Event-Based Middleware for Collaborative Ad Hoc Applications

A thesis submitted to the
University of Dublin, Trinity College,
in fulfilment of the requirements for the degree of
Doctor of Philosophy (Computer Science).

René Meier

Distributed Systems Group,
Department of Computer Science,
Trinity College, University of Dublin.

September 2003.

DECLARATION

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this or any other University, and that, unless otherwise stated, it is entirely my own work.

René Meier,

30th September 2003.

PERMISSION TO LEND AND/OR COPY

I, the undersigned, agree that the Trinity College Library may lend and/or copy this thesis upon request.

René Meier,

30th September 2003.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my supervisor, Prof. Vinny Cahill, for supporting me throughout my PhD studies. His support, both in terms of encouragement as well as technical have provided the best possible foundation for this thesis and are very much appreciated. Many thanks also to Prof. Paddy Nixon for advising me during the early stage of my studies and for helping to initially formulate the basis for this research.

I am also thankful to all the other academics in the Distributed Systems Group, Dr. Siobhán Clarke, Dr. Christian Jensen, and Dr. Simon Dobson. Their technical expertise has been invaluable.

Further thanks go to Brendan O'Brien and Dublin City Council for providing the traffic data that made the evaluation of this research possible.

I would like to specially thank all former and current members of the Anois and the CORTEX project teams, especially Raymond Cunningham for always willingly lending a hand, as well as Dr. Stefan Weber, Dr. Marco Killijian, Dr. Tilman Schaefer, Mads Haahr, Jim Dowling, Peter Barron, Tim Walsh, and Sotirios Terzis, for fruitful collaboration and good spirit.

Finally, I would like to thank my whole family and Louise for their love and constant support and of course, all my friends, especially those in Ireland and in Switzerland.

“Now gentlemen, let us do something which the world may talk of hereafter.”

- Cuthbert Collingwood, 21st October 1805.

ABSTRACT

Middleware supporting event-based communication is widely recognised as being well suited to interconnecting the components of mobile applications since it naturally accommodates a dynamically changing population of components and the dynamic reconfiguration of the connections between them. Existing research on event-based middleware for wireless networks has mainly focussed on accommodating nomadic applications using infrastructure networks while relatively little work has been done to address the distinct requirements associated with supporting collaborative applications, especially those that use ad hoc networks.

Traditionally, event-based middleware employs logically centralised or intermediate components to implement key properties of the middleware. Application components may utilise centralised lookup and naming services to discover peers in order to communicate with them. Intermediate components may be used to route event notifications from producers to consumers and to apply event notification filters. Moreover, they may be used to enforce non-functional attributes, such as event notification delivery order and priority. The central problem with this approach arises with increasing system scale as such middleware components may become a liability due to availability and bandwidth limitations.

Centralised or intermediate middleware components are typically hosted by physical machines that are part of a designated service infrastructure in order to ensure that they are always accessible to application components. The disadvantage of exploiting such an infrastructure is that its installation and maintenance requires substantial resources while limiting communication between components to the geographical areas in which the infrastructure has previously been made available.

A similar approach is generally used when designing event-based middleware for wireless communication using infrastructure networks. Designated middleware components can be hosted naturally by parts of the network infrastructure, such as access points. However, alternative approaches have to be adopted by event-based middleware for ad hoc networks since ad hoc networks allow application components to communicate and collaborate in a spontaneous manner without the aid of a separate service infrastructure.

The main contribution of the work described in this thesis is the design of an event-based middleware for wireless, ad-hoc environments addressing these problems. We have designed an inherently distributed event-based middleware architecture that does not rely on

the presence of any infrastructure or on centralised or intermediate components. Our approach allows the components comprising collaborative applications to come together anywhere and at any time in order to interact through wireless communication using ad hoc networks without depending on a previously installed service infrastructure.

This design supports distributed approaches to discovering peers and to filtering event notifications. Filters may be applied to a range of functional and non-function attributes associated with event notifications including subject, content, and context. Combining event notification filters increases the filtering precision allowing a component to subscribe to the subset of event notifications in which it is interested using multiple criteria, such as meaning, time, location, and quality of service. In particular, event notification filters may be used to define geographical areas within which certain event notifications are valid; hence delivering event notifications at the specific location where they are relevant. Such geographical scopes represent a natural way to identify event notifications of interest for mobile components.

In addition, we have discovered decentralised techniques that improve system scalability for mobile computing applications comprising of large numbers of interconnected components distributed over large geographical areas as well as the predictability of the event service. Event notification filtering in general and our approach of combining filters in particular improves system scalability by limiting the forwarding of event notifications. Moreover, limiting event notification forwarding and bounding the event notification dissemination scope improves the predictable behaviour of the middleware.

A further contribution of this thesis is a taxonomy of distributed event-based programming systems. The taxonomy is structured as a hierarchy of the fundamental properties of a distributed event-based programming system and may be used as a framework to describe an event system according to a variety of criteria including its event, organisation, and interaction models.

In order to validate our contributions, we present an evaluation of a number of mobile application scenarios implemented using middleware that employs our inherently distributed event architecture and our decentralised techniques for improving system scalability and service predictability. The evaluation demonstrates how components of mobile applications can be interconnected through wireless communication and ad hoc networks as well as how our approach to event notification filtering increases filtering precision, improves system scalability, and enhances the predictability of the middleware. In addition, we apply our taxonomy of distributed event-based programming systems to our middleware as well as a selection of other event systems comparing their approaches to providing middleware properties.

RELATED PUBLICATIONS

Journal Papers:

- R. Meier, "Communication Paradigms for Mobile Computing," *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 6, pp. 56-58, 2002.

Refereed Conference and Workshop Papers:

- R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," in *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03), LNCS 2893*. Paris, France: Springer-Verlag Heidelberg, Germany, 2003, pp. 285-296.
- R. Meier and V. Cahill, "Location-Aware Event-Based Middleware: A paradigm for Collaborative Mobile Applications?," presented at the 8th CaberNet Radicals Workshop, Ajaccio, Corsica, France, 2003.
- R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 639-644.
- R. Meier and V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 585-588.
- M. O. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahill, "Towards Group Communication for Mobile Participants," in *Proceedings of Principles of Mobile Computing (POMC'2001)*. Newport, Rhode Island, USA, 2001, pp. 75-82.
- R. Meier, M. O. Killijian, R. Cunningham, and V. Cahill, "Towards Proximity Group Communication," presented at Advanced Topic Workshop on Middleware for Mobile Computing (IFIP/ACM Middleware 2001), Heidelberg, Germany, 2001.
- M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul, "Filtering and Scalability in the ECO Distributed Event Model," in *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (ICSE/PDSE2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 83-95.

CONTENTS

CHAPTER 1: INTRODUCTION 1

1.1 Mobile Computing Environments 2

1.2 Nomadic and Collaborative Applications 2

1.3 Event-Based Middleware and Mobile Computing 4

1.4 Aims and Objectives 4

1.5 STEAM 5

1.6 Contribution of this Thesis 6

1.7 Organisation of this Thesis 8

CHAPTER 2: DISTRIBUTED EVENT MODELS 10

2.1 The Event-Based Communication Model 11

2.1.1 The Participants 11

2.1.2 Characteristics of Event-Based Communication 12

2.2 Functional Design Issues 14

2.2.1 Typed Events 15

2.2.2 Event Filtering 15

2.2.3 Announcing Events 16

2.2.4 Subscribing to Events 16

2.2.5 Delivery Semantics 17

2.2.6 Subject-Based Event Models 17

2.2.7 Content-Based Event Models 18

2.3 Non-Functional Design Issues 18

2.3.1 Quality of Service 19

2.3.2 Real-Time 20

2.3.3 Scalability 21

2.4 OMG CORBA Event Models 22

2.4.1 CORBA Event Service 23

2.4.2 CORBA Notification Service 26

2.4.3 Summary 31

2.5 OMG CORBA Event Model Extensions 32

2.5.1 TAO Real-Time Event Service 32

2.5.2 CONCHA 35

2.5.3 Summary 35
2.6 JAVA Event Models 36
2.6.1 Java AWT Delegation Event Model 36
2.6.2 Java Distributed Event Model 39
2.6.3 Summary 41
2.7 CEA 42
2.7.1 Cambridge Event Architecture 42
2.7.2 Summary 45
2.8 ECO 46
2.8.1 ECO Architecture 46
2.8.2 Summary 49
2.9 JEDI 49
2.9.1 JEDI Architecture 50
2.9.2 Summary 52
2.10 SIENA 53
2.10.1 SIENA Architecture 53
2.10.2 Summary 58
2.11 Elvin 59
2.11.1 Elvin4 Architecture 59
2.11.2 Summary 61
2.12 Perspectives 62

CHAPTER 3: TAXONOMY OF DISTRIBUTED EVENT-BASED PROGRAMMING SYSTEMS 65

3.1 Introduction 65
3.1.1 Exploiting the Taxonomy 66
3.1.2 Related Work 66
3.1.3 Interpreting the Taxonomy 67
3.2 The Taxonomy 68
3.2.1 Event Model 69
3.2.2 Event Service 74
3.3 Classification of Event Systems 100
3.4 Summary 101

CHAPTER 4: THE STEAM EVENT-BASED MIDDLEWARE FOR COLLABORATIVE APPLICATIONS 103

4.1 Proximity-Based Event Notifications 103
4.1.1 Event-Based Middleware for Collaborative Applications 103

4.1.2 Programming Model ... 106
4.1.3 Architecture ... 107
4.1.4 Summary ... 108
4.2 The STEAM Event Model ... 109
4.2.1 Supporting Mobility ... 110
4.2.2 Proximities ... 112
4.2.3 Event Types ... 114
4.3 Event Notification Filtering in STEAM ... 122
4.3.1 Exploiting Distributed Event Notification Filters ... 122
4.3.2 Applying Distributed Event Notification Filters ... 124
4.3.3 Defining Distributed Event Notification Filters ... 126
4.4 Communications Architecture ... 128
4.4.1 Exploiting Proximity Groups ... 128
4.4.2 Locating Proximity Groups ... 130
4.4.3 Mapping to Proximity Groups ... 131
4.4.4 Mapping to Ad Hoc Networks ... 133
4.4.5 Routing Event Notifications ... 135
4.5 Interface Functions ... 136
4.5.1 Delivering Event Notifications ... 136
4.5.2 STEAM Application Programming Interface ... 137
4.6 Discussion ... 137
4.6.1 Mobility ... 138
4.6.2 Scalability ... 138

CHAPTER 5: STEAM ARCHITECTURE AND ALGORITHMS ... 140

5.1 STEAM Architecture ... 140
5.1.1 Overview ... 140
5.1.2 Distribution ... 142
5.1.3 Proximity-Based Event Notification Management ... 143
5.2 The STEAM Event Service ... 146
5.2.1 Event Notifications and Event Types ... 146
5.2.2 Event Notification Filters ... 149
5.2.3 Repositories ... 151
5.2.4 Addressing Scheme ... 151
5.3 Discovering Proximities ... 155
5.3.1 Announcing Event Types and Proximity Filters ... 155
5.3.2 Discovering Event Types and Proximity Filters ... 157

5.3.3 Maintaining Event Types and Proximity Filters ... 158
5.3.4 Discovery Range ... 158
5.4 Disseminating Event Notifications and Announcements ... 159
5.4.1 Proximity-Based Multicast Groups ... 159
5.4.2 Routing Messages ... 160
5.5 Exploiting Geographical Location Information ... 163
5.6 Summary ... 164

CHAPTER 6: EVALUATION ... 166
6.1 Disseminating Event Notifications ... 167
6.1.1 The Application Scenarios ... 167
6.1.2 The Experiment ... 169
6.1.3 Results and Analysis ... 172
6.2 Discovering Proximities ... 176
6.2.1 The Application Scenarios ... 177
6.2.2 The Experiment ... 177
6.2.3 Results and Analysis ... 178
6.3 Event Notification Filtering Precision ... 180
6.3.1 The Intersection Scenario ... 180
6.3.2 The Experiment ... 182
6.3.3 Results and Analysis ... 184
6.4 Raising and Delivering Event Notifications ... 186
6.4.1 The Experiment ... 187
6.4.2 Results and Analysis ... 188
6.5 Summary ... 190

CHAPTER 7: CONCLUSIONS AND FUTURE WORK ... 193
7.1 Achievements ... 193
7.2 Open Research Issues ... 197
7.3 Conclusion ... 198

APPENDIX: SUMMARY OF TAXONOMY PROPERTIES ... 199

BIBLIOGRAPHY ... 205

LIST OF FIGURES

Figure 1.1. Accident warning application scenario. 3
Figure 2.1. Client/server and event-based communication model. 12
Figure 2.2. Announcing and subscribing to events. 16
Figure 2.3. The CORBA event channel. 24
Figure 2.4. Communication styles. 24
Figure 2.5. CORBA event channel interfaces. 25
Figure 2.6. The CORBA notification channel. 27
Figure 2.7. The general structure of structured event data. 30
Figure 2.8. Notification Service event data filter object. 31
Figure 2.9. The TAO RT event channel. 33
Figure 2.10. The CONCHA event channel. 35
Figure 2.11. EventListener interface and EventObject class. 37
Figure 2.12. The Java Delegation Event Model. 38
Figure 2.13. Using event adapters in the Delegation Event Model. 39
Figure 2.14. RemoteEventListener interface and RemoteEvent class. 40
Figure 2.15. Using distributed event adapters in the Distributed Event Model. 41
Figure 2.16. An event source object of an active badge system. 42
Figure 2.17. An event client object of an active badge system. 43
Figure 2.18. Overview of the Cambridge Event Architecture. 43
Figure 2.19. Defining event filters using parameter templates. 44
Figure 2.20. Defining composite event filters using multiple parameter templates. 45
Figure 2.21. The application programming interface defined by ECO. 47
Figure 2.22. Overlapping zones. 48
Figure 2.23. Nested zones. 48
Figure 2.24. Targeting zones. 49
Figure 2.25. Overview of the JEDI architecture. 50
Figure 2.26. JEDI dispatching server topology. 51
Figure 2.27. JEDI events. 51
Figure 2.28. Overview of the SIENA architecture. 54
Figure 2.29. SIENA application programming interface. 54
Figure 2.30. SIENA event server topologies. 56

Figure 2.31. SIENA event. 57
Figure 2.32. SIENA event filter. 57
Figure 2.33. SIENA event pattern. 57
Figure 2.34. Overview of the Elvin4 architecture. 59
Figure 2.35. The Elvin4 security framework. 61
Figure 3.1. Taxonomy legend. 67
Figure 3.2. The root of the taxonomy. 69
Figure 3.3. Event system overview. 69
Figure 3.4. Event model categories. 70
Figure 3.5. A producer and a consumer application using the peer to peer-based Java distributed event model. 71
Figure 3.6. A producer and a consumer application using the mediator-based CORBA event model. 72
Figure 3.7. A producer and a consumer application using the implicit Direct CEA. 73
Figure 3.8. The event service. 75
Figure 3.9. Event service organisation. 76
Figure 3.10. Centralised event service with collocated middleware. 77
Figure 3.11. Distributed event service with collocated middleware. 77
Figure 3.12. Centralised event service with separated single middleware. 78
Figure 3.13. Distributed event service with separated single middleware. 78
Figure 3.14. Centralised event service with separated multiple middleware. 78
Figure 3.15. Distributed event service with separated multiple middleware. 78
Figure 3.16. Event service interaction model. 80
Figure 3.17. No intermediate. 81
Figure 3.18. Distributed intermediate. 81
Figure 3.19. Single centralised intermediate. 82
Figure 3.20. Multiple centralised intermediate. 82
Figure 3.21. Event service features. 85
Figure 3.22. Event propagation model. 86
Figure 3.23. Event type. 87
Figure 3.24. Event filter. 88
Figure 3.25. Event filter location. 89
Figure 3.26. Event filter definition. 90
Figure 3.27. Event filter implementation. 90
Figure 3.28. Event filter evaluation. 91
Figure 3.29. Event filter expressive power. 92

Figure 3.30. Mobility support. 93
Figure 3.31. Composite events. 94
Figure 3.32. Quality of service. 96
Figure 3.33. Ordering. 97
Figure 3.34. Security. 98
Figure 3.35. Failure mode. 99
Figure 4.1. STEAM event model. 109
Figure 4.2. Proximity definition. 110
Figure 4.3. Stationary proximity 111
Figure 4.4. Mobile proximity. 111
Figure 4.5. Disseminating event notifications using overlapping proximities. 114
Figure 4.6. Moving between proximities 115
Figure 4.7. STEAM event type and instance definition. 116
Figure 4.8. Dependencies between non-functional attributes. 119
Figure 4.9. Traffic light event type and event instance example. 122
Figure 4.10. Matching distributed event notification filters. 125
Figure 4.11. Example of the subject and content of an event notification. 126
Figure 4.12. Defining a subject filter. 127
Figure 4.13. Filter term definition. 127
Figure 4.14. Defining a conjunctive content filter. 127
Figure 4.15. Defining a stationary proximity filter. 128
Figure 4.16. Single-hop event dissemination. 133
Figure 4.17. Multi-hop event dissemination. 134
Figure 4.18. Network partitions in multi-hop event dissemination 135
Figure 4.19. Delivering event notifications in STEAM. 136
Figure 4.20. The application programming interface of STEAM. 137
Figure 5.1. The architecture of the STEAM middleware. 141
Figure 5.2. Mobile devices hosting STEAM middleware. 142
Figure 5.3. Announcing and discovering proximity-based event notifications. 143
Figure 5.4. Subscribing to proximity-based event notifications. 144
Figure 5.5. Raising proximity-based event notifications. 145
Figure 5.6. Delivering proximity-based event notifications. 146
Figure 5.7. Defining the structure of an event type. 147
Figure 5.8. Instantiating an event notification. 148
Figure 5.9. Content filter term operators. 149
Figure 5.10. Instantiating stationary and mobile proximity filters. 150

Figure 5.11. Maintaining announcement and subscription information. 151
Figure 5.12. Computing group identifiers from event type and proximity pairs..................... 153
Figure 5.13. A producer generating event notifications. 154
Figure 5.14. A consumer receiving event notifications. 154
Figure 5.15. Announcing event type and proximity filter pairs. 156
Figure 5.16. Discovering event type and proximity filter pairs. 157
Figure 5.17. The programming interface of the proximity-based group communication service. 160
Figure 5.18. Receiving single-hop and multi-hop messages. 162
Figure 6.1. Application scenario overview.. 169
Figure 6.2. Cost of event notification dissemination in scenario (A_1) as a function of proximity range. 172
Figure 6.3. Cost of event notification dissemination in scenarios (B_1) and (C) as a function of proximity range. 173
Figure 6.4. Cost of event notification dissemination in scenarios (D) and (E) as a function of proximity range. 174
Figure 6.5. Cost of event notification dissemination in scenario (A_2) for a saturation of 120 as a function of subscriber speed. 174
Figure 6.6. Cost of event notification dissemination in scenario (B_2) for a saturation of 120 as a function of producer speed. 175
Figure 6.7. Event notification dissemination cost reduction due to gossiping....................... 175
Figure 6.8. Fraction of gossiping consumers losing event notifications in scenarios with a saturation of 60.. 176
Figure 6.9. Cost of proximity discovery in scenarios (C) and (D) as a function of the discovery range. 178
Figure 6.10. Discovery coverage ratio in scenarios (C) and (D) as a function of the discovery range. 179
Figure 6.11. The North Circular Road and Prussia Street intersection................................ 181
Figure 6.12. Example phase scheme for a two-approach intersection................................ 182
Figure 6.13. Modelling the intersection. 182
Figure 6.14. The number of event notifications delivered on each of the lanes. 185
Figure 6.15. The average number of events delivered by individual vehicles on each lane. 185
Figure 6.16. Precision of event notification filtering for various filter combinations. 186
Figure 6.17. Definition of the “Performance” event type. 187
Figure 6.18. Definition of the conjunctive content filter applied to the disseminated event notifications.. 188
Figure 6.19. Latency of a producer raising an event notification as a function of the number of announced event notification types and the number of subscribers.................................... 189
Figure 6.20. Latency of a consumer delivering an event notification as a function of the number of subscriptions to other event notification types. 190

LIST OF TABLES

Table 2.1. Event model terminology. 12
Table 2.2. The administrative properties of notification channels. 28
Table 2.3. Notification Service QoS properties. 29
Table 3.1. Categorisation of event systems. 101
Table 4.1. Defining functional attributes. 116
Table 4.2. Defining non-functional attributes. 117
Table 4.3. Classification of attributes. 121
Table 5.1. Maintaining proximity filters. 158
Table 6.1. Description of the application scenarios. 168
Table 6.2. The first three records of the intersection data. 181
Table 6.3. The configuration of the experiment. 183
Table 6.4. The configuration of the runs. 184

CHAPTER 1: INTRODUCTION

The term middleware applies to a layer of software whose purpose is to mask the heterogeneity of the underlying distributed system and to provide a convenient programming model to application programmers [1]. Modern middleware platforms, such as CORBA [2] and Java Remote Method Invocation (RMI) [3], have mainly focussed on supporting a programming model based on the traditional client/server communication model, which typically provides synchronous, point-to-point communication between client and server. This communication model requires clients and servers to have some knowledge of each other, i.e., they must be aware of each other's addresses, in order for a client to be able to send a service request and for a server to send the corresponding response.

Emerging mobile and ubiquitous computing applications typically comprise large numbers of interconnected components distributed over large geographical areas. Middleware supporting such applications must deal with the increased complexity that comes with such scale and the geographical dispersion of components as well as the spontaneously changing connections between components that may be either static or mobile.

The synchronous nature of middleware based on the client/server paradigm results in stable, relatively long lasting connections between communicating participants in which the server blocks an interaction while computing its response. However, mobile and ubiquitous applications that involve unanticipated interactions between loosely coupled components require a different approach to application component integration, due to the dynamic character of such interactions.

The event-based communication model, a paradigm for middleware that asynchronously [4] interconnects the components that comprise an application, is widely recognised as being well suited to addressing the requirements of mobile applications [5-8]. It avoids centralised control and requires a less tightly coupled communication relationship between components compared to the client/server communication model.

Event-based middleware allows one component to react to a change occurring in another component. A source propagates an event notification while a potential receiver determines its interest. Event-based middleware naturally accommodates a dynamically changing

population of components and is particularly useful in wireless networks, where communication relationships amongst heterogeneous application components are dynamically reconfigured.

1.1 Mobile Computing Environments

Mobile computing environments can use either the infrastructure or the ad hoc network model for wireless communication [9]. The infrastructure model exploits access points to enable communication among the mobile application components under its direct control and to coordinate their transmissions analogous to the base station in a cellular communications network. Access points may be connected to a fixed network, such as a company intranet or the Internet, and act as portals allowing the components under their control to connect to the fixed network.

In contrast, the ad hoc network model allows application components to communicate with each other without the aid of access points or a fixed network. Any application component can establish a direct (single-hop) communication relationship with any other application component within its transmission reach without having to channel the transmission through an access point. An application component may communicate with components potentially located beyond its transmission reach through multi-hop communication where intermediate application components forward a transmission towards its destination. Hence, the ad hoc network model allows application components to communicate and collaborate in a spontaneous manner at any time and without restrictions, except for connectivity limitations, in the absence of a conventional fixed network.

The term mobile device is used to refer to portable computing devices, such as notebook computers and handheld devices. A mobile device may be capable of wireless networking, thus allowing its application components to interact with components hosted by other mobile devices through wireless communication while moving in a mobile computing environment.

1.2 Nomadic and Collaborative Applications

We characterise applications in the space of mobile computing as either nomadic or collaborative. Nomadic applications are characterized by the fact that mobile nodes make use of the wireless network primarily to connect to a fixed network infrastructure, such as the Internet, but may suffer periods of disconnection while moving between points of connectivity. Consequently, the main goal of middleware accommodating nomadic applications has been

to handle disconnection while nodes migrate from one designated access gateway to another. This implies that such middleware has focused on providing a means to cache and synchronise relevant information on behalf of a disconnected node and to forward it via the new access gateway upon reconnection. Nomadic applications typically employ infrastructure networks when connecting to the fixed backbone network. An example of a nomadic application might include a news ticker hosted by a handheld device accessing an Internet-based breaking news service through a wireless connection. Such a ticker application might be disconnected from the news server while the user travels from her home to her office.

In contrast, collaborative applications can be characterized by the fact that mobile nodes use the wireless network to interact with other mobile nodes that have come together at some common location. Collaborative nodes migrate within some area, establish associations with other nodes dynamically, and typically group into formations of nodes that have a common goal. The members of such a group may migrate together, similar to a fleet of vehicles or the participants of a guided tour on an excursion. Although these applications may use infrastructure networks, they will often use ad hoc networks since they are immediately deployable in arbitrary environments and support communication without the need for a separate infrastructure. This collaborative style of application may be useful in the ubiquitous [10] and sentient computing [11] domain allowing loosely coupled, highly mobile components to communicate and collaborate in a spontaneous manner. Such applications might include scenarios in which a crashed car disseminates an accident warning to approaching vehicles and players in augmented reality games being interested in the status of game objects or indeed other players residing at their current location. Figure 1.1 further illustrates the accident warning application scenario. The car shown in the centre of Figure 1.1 has crashed and as a result establishes associations with vehicles that are currently near the location of the accident. The crashed car may subsequently establish associations with other vehicles that approach the accident site as well as cancel associations with vehicles that have passed the site.

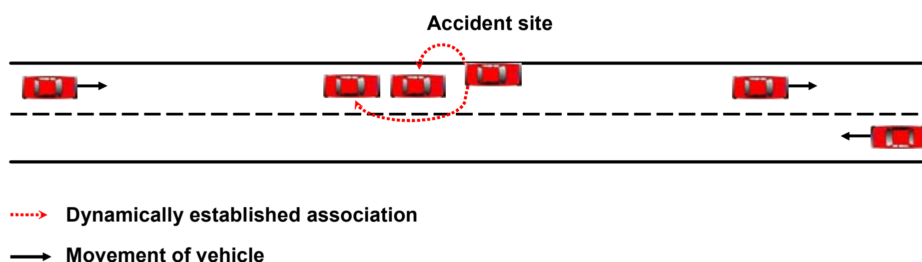


Figure 1.1. Accident warning application scenario.

1.3 Event-Based Middleware and Mobile Computing

Middleware services supporting event-based communication have been developed by both industry [12, 13] and academia [4, 5, 14, 15]. Most of these assume that the components comprising an application are stationary and that a fixed network infrastructure is available. Existing research on event-based middleware for mobile applications has mainly concentrated on supporting nomadic applications and wireless data communication based on infrastructure networks [4-6, 16, 17], assuming the availability of a fixed service infrastructure. Intermediate middleware components, which typically implement the mechanisms for handling disconnection as well as for enforcing many middleware properties, are naturally hosted by parts of the designated service infrastructure.

Relatively little work has been done to address the distinct requirements associated with event-based middleware accommodating collaborative applications, especially those that use ad hoc networks. Such applications may lack any service infrastructure and therefore can not rely on the aid of access points when routing event notifications or when discovering peers. Moreover, event notifications can not depend on intermediate components applying event notification filters or enforcing non-functional policies, such as event notification delivery order and priority. Consequently, alternative approaches have to be adopted by event-based middleware for collaborative applications in order to provide the desired middleware properties in the absence of a designated service infrastructure as well as centralised or intermediate components typically hosted by such an infrastructure.

1.4 Aims and Objectives

The general objective of this thesis is to design an event-based middleware architecture for collaborative applications, especially for those that use ad hoc networks. The middleware should allow collaborative application components to interact through wireless data communication while featuring a number of desirable functional and non-functional properties. From the application programmer's point of view, the design of such middleware should provide for:

- Ad hoc networks. The middleware should support an event-based programming model allowing collaborative application components, with significant variations in speed from stationary to highly mobile, to come together at a certain location and then to communicate and collaborate through wireless connections using ad hoc networks.

- Inherently distributed architecture. The middleware should exclusively use the same physical machine as the components that comprise the collaborative application and not rely on the presence of a designated service infrastructure.
- Event notification filtering precision. The middleware should support a range of event notification filters that may be applied to various attributes of event notifications including subject, content, and context, such as geographical location. Moreover, it should allow a subscriber to combine event notification filters in order to describe the exact subset of event notifications in which it is interested exploiting multiple criteria, such as meaning, time, location, and quality of service.
- System scalability. A system exploiting event-based middleware for collaborative applications should be able to easily cope with a large, dynamically changing population of mobile components distributed over a large geographical area and the resulting dynamic reconfiguration of the connections between the components.

These middleware properties are not orthogonal and hence can not be addressed independently. We therefore have to find a good compromise when designing the middleware supporting the properties described above. For example, an inherently distributed architecture implies absence of components that have global knowledge. Such components are traditionally exploited when applying event notification filters. Consequently, a new technique for applying event notification filters in a distributed manner is needed.

1.5 STEAM

We envisage event-based middleware for collaborative applications being used in various areas including indoor and outdoor smart environments, augmented reality, and traffic management. In these application scenarios, components may represent mobile objects ranging from robots and cars to buses and fire engines, as well as objects with a fixed location, such as office appliances, information points, traffic signals, and traffic lights.

This thesis argues that there are applications in which collaborative components are more likely to interact once they are in close proximity. Components within close vicinity may communicate using the middleware in order to exchange information on the status of a door, the theme of a museum, or the current traffic situation. In a traffic management application scenario, a traffic signal may propagate an alteration to the speed limit due to changing road conditions to approaching vehicles. Another example scenario may involve an ambulance disseminating its location to nearby vehicles in order for them to yield the right of way.

We present an implementation of our middleware architecture for collaborative applications, called STEAM (Scalable Timed Events And Mobility) [18, 19]. STEAM is intended for applications that include a large number of highly mobile, collaborative application components typically distributed over a large geographical area. Unanticipated interaction between nearby components is supported enabling a component to dynamically establish connections to other components within its current vicinity. This allows components representing real world objects currently residing within the same geographical area to deliver events at the location where they are relevant.

1.6 Contribution of this Thesis

Within the context of this thesis we have designed and implemented an event-based middleware for collaborative applications. Our work has focused on designing an event-based middleware that is especially suited for those collaborative applications that use ad hoc networks while supporting a number of middleware features typically desired by application programmers in this domain.

Traditionally, event-based middleware employs logically centralised or intermediate components to implement key properties of the middleware. Application components may utilise centralised lookup and naming services to discover peers in order to communicate with them. Intermediate components may be used to route event notifications from producers to consumers and to apply event notification filters. Moreover, they may enforce non-functional attributes, such as event notification delivery order and priority. However, the central problem with this approach arises with increasing system scale as such middleware components may become a liability due to availability and bandwidth limitations.

Centralised or intermediate middleware components are typically hosted by physical machines that are part of a designated service infrastructure in order to ensure that they are always accessible to application components. The disadvantage of exploiting such an infrastructure is that its installation and maintenance requires substantial resources while limiting communication between components to the geographical areas in which the infrastructure has previously been made available.

A similar approach is generally used when designing event-based middleware supporting nomadic applications. Designated middleware components can be hosted naturally by parts of the network infrastructure, such as access gateways. However, such an approach is inadequate for event-based middleware for collaborative applications, especially for those using ad hoc networks, due to the lack of any infrastructure.

The main challenge of our work has been to design event-based middleware supporting collaborative applications that adopts alternative approaches addressing these problems without the aid of a separate service infrastructure while avoiding centralised and intermediate components. A further challenge has been to provide event notification filtering with high precision allowing a component to use multiple functional and non-function criteria when identifying event notifications of interest. The final challenge of this thesis has been to develop decentralised techniques that improve system scalability for applications composed of large numbers of interconnected mobile (and static) components distributed over large geographical areas.

The main contribution of the work described in this thesis is the design of an event-based middleware for collaborative applications addressing these challenges. Consequently, our middleware has a number of important differences from other event services:

- Mobility support. Collaborative application components interact through wireless communication utilising the ad hoc network model without the aid of access points or connections to a conventional fixed network. Our design accommodates a changing pool of collaborative application components coming together at a location and supports spontaneous communication between these components without preceding infrastructure deployment.
- Inherently distributed architecture. The middleware is exclusively collocated with the collaborative application components and does not depend on centralised or intermediate components. Decentralised techniques for discovering peers and for filtering of event notifications are supported. This is beneficial as it avoids components typically hosted by a designated service infrastructure that may become communication bottlenecks with increasing system scale.
- Location-aware application components. Geographical location information is provided by a location service, which is essential for geographical scoping of event notifications used to deliver event notifications at the specific location where they are relevant.

- Distributed event notification filtering. Event notifications may be filtered at both the producer and the consumer side or may be filtered implicitly. Filters may be applied to a range of functional and non-function attributes associated with an event notification including subject, content, and context, such as geographical location. Combining distributed event notification filters is beneficial to the precision of filtering allowing a component to define the subset of event notifications in which it is interested using multiple criteria, such as meaning, time, location, and quality of service. Event notification filtering in general and our approach of combining filters in particular improves system scalability by limiting forwarding of event notifications.
- Geographical scoping of event propagation. To support ad hoc networks, event notification filters may be used to define geographical areas within which certain event notifications are valid, hence bounding the geographical scope within which these event notifications are propagated. Such geographical scopes represent a natural way to identify event notifications of interest for mobile components. Geographical scoping is essentially filtering of event notifications using the space criteria and consequently increases system scalability further. Bounding the dissemination range of event notifications improves the predictable behaviour of the middleware.
- Non-functional application requirements. The middleware architecture and the decentralised techniques for peer discovery and event notification filtering have been designed to improve system scalability and the predictability of the filter engine for event notifications. Other non-functional application requirements, such as event notification delivery order and priority, may be associated with either a specific event notification or a group of event notifications using attributes.

A further contribution of this thesis is a taxonomy of distributed event-based programming systems. The taxonomy is structured as a hierarchy of the fundamental properties of a distributed event-based programming system and may be used as a framework to describe an event system according to a variety of criteria including its event, organisation, and interaction models. The taxonomy has been applied to our middleware as well as a selection of other event systems to compare their middleware properties.

1.7 Organisation of this Thesis

After this introduction to event-based middleware, mobile computing environments, and mobile application styles, we structure the remainder of this thesis as follows: Chapter 2 introduces the terminology and the characteristics of the event-based communication model

and subsequently reviews and examines work related to this thesis. In chapter 3, we present our taxonomy of distributed event-based programming systems and classify a selection of event models comparing their properties. Chapter 4 describes the STEAM event model and the rationale for its design. Chapter 5 presents a prototypical implementation of the STEAM event model. In chapter 6, we validate our work by presenting an evaluation of a number of collaborative application scenarios using the STEAM middleware. This chapter demonstrates how components of collaborative applications can be interconnected through wireless communication and ad hoc networks as well as how our approach to event notification filtering increases filtering precision, improves system scalability, and enhances the predictability of the filter engine. Finally, chapter 7 concludes this thesis by summarising the presented work and outlining issues that remain open for future work.

CHAPTER 2: DISTRIBUTED EVENT MODELS

The first part of this chapter introduces the characteristics, related terminology, and main concepts of distributed event-based programming models as well as the issues that arise when designing distributed event models. The subsequent sections then review a number of event-based communication models related to the work described in this thesis, which have been selected based on their popularity and according to the features they support. We conclude this chapter by discussing the mobility support provided by the event-based communication models surveyed.

The review is concerned with middleware that provides a communication model based on the event paradigm. An event-based middleware, which is also known as an event service, is characterised by its architecture and the features that it supports. The architecture specifies the overall structure of the service as well as the components involved and their inter-relationships. The set of features supported by a specific event service reflects the requirements of the application domain for which it has been designed and hence may vary considerably. We review a number of event-based communication models developed by both industry and academia that have been designed for a range of application areas including large-scale Internet services [20] and mobile computing [21, 22]. As a result, the requirements of these applications cover different scalability and timeliness constraints as well as a range of computing environments including fixed and mobile networks. These event models have been selected firstly according to their relevance to the work presented in this thesis and secondly, to illustrate a wide range of architecture styles, features, and issues that serve as the basis for identifying the properties of the taxonomy of distributed event-based programming systems presented in chapter 3. Hence, this selection of influential event models are reviewed according to their system architecture, the supported programming model, and their functional and non-functional features, especially those related to event filtering, mobility support, and quality of service.

2.1 The Event-Based Communication Model

The general idea behind the use of an event-based communication model is to enable one application component to react to a change in the state of another component. Event-based communication models, or simply event models, are omnipresent in applications ranging from small-scale, centralised to large-scale, highly distributed systems [23]. On one hand, they are exploited to interconnect individual components of applications, for example the components comprising graphical user interfaces [24, 25]. Such graphical components may disseminate user driven and hence sporadic changes to their state to other components of the application that are required to react to these changes. At the other extreme, publishers of stock trading information may utilise a system with an event service to post the latest trading rates to a group of brokers, potentially located in different cities or even countries [26, 27]. Smart environments often employ event-based communication models to interconnect a large number of application components [28] ranging from light and door actuators and sensors to robotic vehicles moving within and between buildings.

2.1.1 The Participants

An event system is an application that uses event-based communication to allow the components that comprise the application to interact using event notifications. Event notifications, or simply events, contain data that represent a change to the state of the sending application component. They are propagated from the generating application components, called the producers, to the receiving application components, called the consumers, which process the events delivered.

In conventional distributed event systems, application components, called entities, are located on a number of physical machines that are interconnected by means of a fixed network infrastructure through which communication takes place. Middleware using event-based communication may support intermediate components. Intermediate components typically route events from producing to consuming entities and are potentially hosted by separate machines that are part of the infrastructure.

Event Model	Source	Sink	Intermediate
STEAM [18, 19, 29]	Producer	Consumer	N/A
CORBA [12, 30]	Supplier	Consumer	Channel
TAO RT CORBA [31]	Supplier	Consumer	Real-time channel
CONCHA [32]	Multicast supplier	Multicast consumer	Channel

Event Model	Source	Sink	Intermediate
Java AWT [24]	Source	Listener	Event adapter
Java Distributed [13]	Generator	Listener	Event adapter
CEA [4, 33]	Source object	Client object	Mediator
ECO [14, 34]	Object	Object	N/A
JEDI [5]	Active object	Active object	Dispatching server
SIENA [15]	Object of interest	Interested party	Event server
Elvin [17, 35].	Producer	Consumer	Server

Table 2.1. Event model terminology.

There is no generally accepted standard for the terminology used for event-based communication. As a result, the event models reviewed in this thesis use a variety of alternative terminology, which is summarised in Table 2.1, when referring to event producer (source), consumer (sink), and intermediate. Although some event models use alternative terminology for event notification, such as event object, event message, and event data, Table 2.1 omits these as the term “event” is widely accepted.

2.1.2 Characteristics of Event-Based Communication

The traditional client/server computing [36] model allows application components to behave as service consumers, called clients, and service providers, called servers. A client/server relationship is established between two application components when one component acting as a client initiates a service request to another component acting as a server that is capable of responding to the service request.

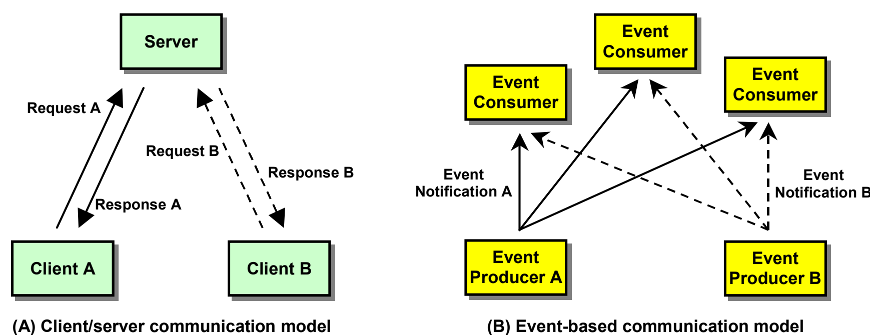


Figure 2.1. Client/server and event-based communication model.

Figure 2.1(A) illustrates request/response interactions between clients and a server, which often reside on separate physical machines and communicate through a network connection. This computing model essentially provides synchronous, one-to-one communication between client and server and requires clients and servers to have some knowledge of each other. In order to be able to send a request, a client needs to know the name and the location of the server and the server needs to know the name and the location of the client when responding. In contrast, computing models based on the event paradigm, as shown in Figure 2.1(B), require a less tightly coupled communication relationship between application components acting as either event producer or consumer, allowing them to interact in an asynchronous, anonymous, one-to-many (many-to-many) manner.

Asynchronous Communication

Coulouris et al. [1] have identified asynchronous interaction as a main characteristics of event-based communication. Event producers disseminate event notifications asynchronously to event consumers without having to synchronise. Asynchronous communication prevents slow, temporarily unavailable, or indeed blocked application components from delaying interactions between components. Producers are not required to wait for responses from consumers before proceeding to propagate subsequent event notifications. An event notification destined for a temporary unavailable consumer may be buffered in order to be delivered once the consumer has recovered.

Anonymity

The event-based communication model allows application components to interact anonymously without concern for either the number or the location of the components involved. Event-based middleware typically manages the connections between producers and consumers transparently on behalf of an application. Such middleware implements a level of decoupling that enables producers to disseminate event notifications to consumers without targeting specific destinations and consumers to deliver events without having directly communicated with producers. However, consumers may derive a certain awareness of their producers from the content of the events they receive. For example, a producer may raise events on behalf of a door disseminating the door's status without targeting specific consumers. A consumer receiving these events will be able to determine whether the door is open or closed and in addition, might discover the location of the door.

Anonymous interaction allows producers and consumers to establish communication relationships relatively easily, involving modest initialisation effort compared to the

client/server communication model. Producers and consumers connect to a middleware and may subsequently publish and receive event notifications whereas clients are required to explicitly establish a two-way connection to each specific server with which they intend to interact.

One-to-Many and Many-to-Many Communication

Figure 2.1(B) illustrates how a producer initiates event-based communication by propagating an event notification to a group of consumers. A single producer disseminates specific event notifications to a group of consumers in a one-to-many fashion. Many-to-many communication may be established by a set of collaborating producers disseminating related event notifications to a group of consumers. One-to-many and many-to-many communication may be implemented as a set of unicast interactions [14] sequentially delivering a particular event notification to a group of consumers but are frequently based on multicast protocols [14, 32, 37, 38].

Heterogeneity

Utilising event-based middleware to integrate distributed application components results in loose coupling between these components. Essentially, all that is required for components to interact is for producers to disseminate event notifications and for consumers to recognise event notifications of interest and to provide an interface for receiving them. Hence, event notifications may be used as a means of communication between distributed application components in a heterogeneous system that were not inherently designed to interoperate [1].

2.2 Functional Design Issues

An event service is a middleware component that implements an event model, thereby providing event-based communication to an event system. The functional requirements (a definition can be found below) of such a system are addressed when designing the architecture and the features of its event model.

> *Functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do [39, p.118].*

2.2.1 Typed Events

In general, events generated by a producer are said to be either generic or typed. The information that describes a generic event is a data blob without an explicit structure. Typed events on the other hand, provide a well-defined, explicit, and expressive data structure into which a wide variety of event data can be mapped. The structure of typed events ranges from simple to complex; from a single string to a programming language specific object with an associated set of attributes and methods. However, many event models support typed events that typically have a name and may have an associated set of typed parameters whose specific values describe the specific change to the producer's state.

It may be argued that simple forms of typed events are merely generic and not typed. However, we consider events that have some structure which may be recognised or interpreted by the event model as typed. Significantly, the structure associated with typed events is essential for applying event filters. For example, events that are based on a single string have an explicit data structure to which filters may be applied whereas events enclosing a binary file do not define a recognisable data structure.

2.2.2 Event Filtering

An event system may consist of a potentially large number of producers, all of which can generate events that contain different, application-specific information. As a result, the number of events to be disseminated in an event-based system may be very large.

A particular consumer may only be interested in a subset of the events produced in the system. Event filters provide a means to control the propagation of events. Ideally, filters enable a particular consumer to specify the exact set of events in which it is interested [7]. Supporting a means of event filtering with a high precision minimises (ideally prevents) the delivery of unwanted events to consumers and consequently reduces the utilisation of

communication and computation resources. Essentially, a consumer's event filters are matched against events and only events for which the matching produces a positive result are subsequently delivered to the consumer.

2.2.3 Announcing Events

Producers may indicate their intention to generate events using advertisements. A specific producer may announce the instances of events it intends to raise. Once certain events have been announced, a producer may publish such events until it indicates that it no longer wishes to produce them by cancelling the corresponding advertisement. A producer unannounces the events it ceases to raise. The announcement mechanism is an optional event model feature and thus may not be explicitly supported. However, an event service may exploit the additional information provided by announcements when routing events from producers to consumers [15].

2.2.4 Subscribing to Events

In order to receive events, event consumers have to subscribe to the instances of events in which they are interested. When doing so, consumers are said to register interest in events. Once consumers have subscribed to events, they receive all subsequently disseminated events until they unsubscribe (de-register). Consumers may pass event filters to the event service when subscribing thereby specifying the events of interest. In essence, announcements describe the events generated by producers whereas subscriptions (and the associated event filters) specify the subsets that specific consumers wish to receive.

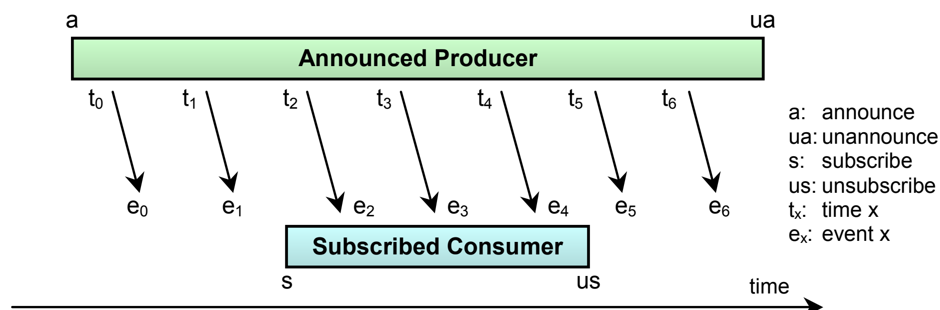


Figure 2.2. Announcing and subscribing to events.

Figure 2.2 summarises the concept of producers announcing and consumers subscribing to events. It depicts an event producer announcing (a) the events it intends to generate, subsequently raising events (e_0 to e_6), and eventually unannouncing (ua) them. Note that no events are published either prior to the announcement or after the unannouncment. A consumer registers interest in these events for a period of time (from s to us) and as a result receives the events disseminated while it has been subscribed. This subscription causes the consumer to deliver events e_2, e_3, and e_4 while prior and subsequent events are disregarded.

2.2.5 Delivery Semantics

Event models support a variety of different delivery guarantees for event notifications. A particular event model may provide several delivery semantics allowing an application to select the semantics that appropriately addresses its requirements. These guarantees naturally relate to the semantics described by the underlying mechanism for sending events to a group of subscribers. For example, exploiting IP multicast for disseminating events will provide a best-effort delivery semantics, which does not guarantee that any subscriber will necessarily receive a specific event. Some applications may have stronger reliability requirements. In order to address these requirements, event models may employ event dissemination protocols that provide at-least-once, at-most-once, or exactly-once delivery semantics.

In addition to providing a certain delivery reliability, event models may enforce delivery semantics disseminating events in a specific order or with some timeliness constraints. A weak delivery order disseminates event in any order whereas a stronger ordering semantics causes events to be delivered in FIFO, causal, or total order. Timeliness constraint are important when delivering events on behalf of real-time applications, such as vehicle control and hospital patient monitoring.

2.2.6 Subject-Based Event Models

Traditionally, event-based communication models support subscription mechanisms based on the subject of an event. In such an approach, each event is classified as belonging to one of a set of subjects, which are also known as topics or channels. Producers label their events with a subject when propagating them and consumers subscribe to a specific subject. A subscriber subsequently receives all events labelled with that subject.

A subject-based event model may support event types that have a name and an associated set of typed parameters. The name of such an event naturally represents its subject and events of the same subject have an identical structure. Hence, the subject of an event allows consumers to subscribe to a specific group of events and identifies their type.

Significantly, subject-based event models allow for an efficient approach to matching events against a large number of subscriptions. For example, a CORBA event channel [30] can be set up as a component handling all events of a particular subject and as a result matches events implicitly. Consumers subscribe to a specific subject by connecting to the corresponding channel. Producers disseminate events by forwarding them to the channel associated with the subject, which then delivers them to its subscribers sequentially.

2.2.7 Content-Based Event Models

As an alternative to subject-based event subscription, event-based communication models may support subscription mechanisms based on the content of an event. The content-based approach allows producers to disseminate events that essentially consist of a set of parameters defining the event information. Consumers subscribe to events by defining a predicate that may test arbitrary parameters of an event. A subscriber subsequently receives all events whose parameters match the subscriber's predicate.

Compared to subject-based approaches, content-based subscription mechanisms provide a more powerful paradigm, allowing consumers to chose filtering criteria along multiple, orthogonal dimensions of the content of events without defining subjects. However, while content-based approaches provide a more expressive subscription mechanism they are difficult to implement [40]. The problem of efficiently matching events against a large number of subscribers has been addressed by the work of Banavar et al. [37] and Opyrchal et al. [38].

2.3 Non-Functional Design Issues

An application using event-based middleware to interconnect its components may have requirements regarding the non-functional behaviour of the system. Similar to functional requirements, non-functional requirements (a definition can be found below) influence the design of the event model implemented by the middleware.

> *Non-functional requirements are constraints on the services or functions offered by the system. They include timing constraints, constraints in the development process, standards and so on. .. Examples are reliability, response time and store occupancy [39, p.119].*

2.3.1 Quality of Service

The non-functional behaviour of an event system may be influenced by a variety of Quality of Service (QoS) constraints. Event-based middleware may support QoS properties that are an integral part of the system and consequently cannot be customised by an application. However, QoS properties may be configurable allowing an application to chose the non-functional behaviour that appropriately addresses its requirements.

Sommerville [39, p.119] identifies reliability, response time and memory management as QoS constraints. Reliability may refer to a number of aspects of an event system including event delivery semantics and connection preservation. Specific event notifications may be guaranteed to be delivered to all subscribers. Information on communication connections between entities and infrastructure may be maintained in order to allow for transparently re-establishing lost connections upon recovery. Event notifications may be stored on behalf of a temporarily unavailable subscriber until they can be delivered when the subscriber recovers. For example, the CORBA Notification Service [12] can be configured to prevent event notification losses. Its event channel may persistently buffer event notifications on behalf of a temporarily unavailable subscriber and subsequently forward these event notifications once the subscriber re-connects. Response time constraints refer to the real-time behaviour of an event system which we discuss separately below. Memory may be managed by allowing an application to impose an upper bound on the sizes of queues as well as on the maximum number of interconnected producing and consuming entities. Various discard policies may be applied to purge expired connection information and stored events. Limiting the use of the available memory is important in any event system, but is essential in those that comprise host machines with strictly limited resources.

2.3.2 Real-Time

Krishna and Shin [41] state that there is no precise, cogent definition of what a real-time system is. They admit that their definition (stated below) raises as many questions as it answers. Notably, it provokes the question as to what “timely” means. However, based on the definitions of real-time systems stated below, we can define the term “timely” as a real-time system hosting tasks that have associated deadlines for their completion.

> *Any system where a timely response by the computer to external stimuli is vital is a real-time system [41, p.1].*
>
> *A real-time system is any information processing activity which has to respond to externally generated input stimuli within a finite and specified time [42, p.2].*
>
> *A system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness [43].*

Real-time systems can be divided into two categories: hard real-time systems and soft real-time systems. Hard real-time (or critical) systems are systems where something “bad” will happen if the output is not delivered in time. Soft real-time (or non-critical) systems are systems where nothing catastrophic happens if some deadlines are missed. Missed deadlines in a soft real-time system will merely result in degradation of performance below what is generally considered acceptable. A deadline can be called hard or soft deadline depending on whether it has been given for a hard or a soft real-time system.

An event system may support real-time guarantees in order to provide deterministic end-to-end behaviour when disseminating events. Although event-based middleware might not be able to guarantee low latency, it may support properties enabling predictions on the behaviour of event propagation. Latency in distributed communication depends on the topology and quality of the underlying network as well as the available bandwidth. This results in “low” latency being relative. Event-based middleware featuring timely event delivery may allow applications to assign priorities to events. As demonstrated by Harrison et al. [31], this allows a dispatcher to pre-empt the delivery of some event in order to deliver an event that has a higher priority. Other real-time event service may support event delivery deadlines,

such as earliest and latest delivery time, and event delivery timeouts. Delivery deadlines describe a time window within which an event is to be delivered and are characteristically expressed using absolute time. Hence, the nodes that comprise an event system exploiting such deadlines require a notion of global time. In contrast, delivery timeouts define a relative duration during which an event is meant to be delivered and thus, do not require synchronised participants.

2.3.3 Scalability

The term scalability (some definitions can be found below) is used to describe the behaviour of a distributed system when changing the number of interconnected participants that form the network. Coulouris et al. [1] state that controlling performance loss and the cost of physical resources as well as preventing software resources from running out and avoiding performance bottlenecks as the challenges presented to the design of a scalable distributed system. A telephone system using up all the available numbers and the Internet (using the 32 bit protocol version 4) running out of computer addresses are examples of systems that show lack of scalability as they run out of software resources. An example of a system in which certain components may become performance bottlenecks is a file server unable to cope with an increasing number of access requests.

> *A scalable distributed system is one that can easily cope with the addition of users and sites, and whose growth involves minimal expense, performance degradation, and administrative complexity [44, p.363].*
>
> *The system and application software should not need to change when the scale of the system increases. .. Rather, as the demand for a resource grows, it should be possible to extend the system to meet it [1, p.20-21].*

Considering these definitions and examples, it can be observed that the scale of an event system depends on several factors. The parameters that influence the scale of a distributed event system include:

- The number of event producing and consuming entities, intermediates, and physical machines.

- The number of activities such as announcements, subscriptions, and event communication.

In principle, these parameters are independent, but in practice they are likely to increase simultaneously. For example, an increase in the number of entities is likely to cause the number of activities to rise. In order for an event system to scale well changing one parameter should not cause another factor to become a performance bottleneck. For example, increasing the number of entities should not result in an intermediate becoming overloaded with event messages. In general, it can be observed that the importance of scalability increases with the complexity of a distributed system and indeed a distributed event system.

2.4 OMG CORBA Event Models

The Common Object Request Broker Architecture (CORBA) is an open standard for object management specified by the Object Management Group (OMG). The architecture uses Object Request Brokers (ORBs) as the middleware for application component integration across boundaries such as networks, operating systems, and programming languages. In order to extend the ORB core capabilities, the CORBA 2 specification [2] defines a wide range of general-purpose services including the CORBA Event Service [30]. This service allows the components that comprise an application to interact using event-based communication in addition to the request/response communication model provided by the bare ORB. The main limitation of the CORBA Event Service is its lack of event filtering and QoS capabilities required by applications such as large-scale and real-time services. The OMG addressed these shortcomings in 1996 by issuing a Request For Proposal [45] for defining an extension to its Event Service. A consortium including Borland International, IONA Technologies, IBM Corporation, and Oracle Corporation submitted a revised proposal [46] that was accepted by the OMG at the end of 1998 and resulted in the specification of an extended version of the Event Service called CORBA Notification Service [12].

The event model of the Event Service and the Notification Service are similar in that both exploit mediator components through which event data is disseminated. Both event models can be characterised as extremely general addressing the needs of different business domains and consequently complex due to the large number of interfaces. However, they are not identical since the Notification Service facilitates additional functionality.

2.4.1 CORBA Event Service

The CORBA Event Service supports an event model that defines two roles for application components. The role of a supplier object producing event data and the role of a consumer object processing event data. Suppliers and consumers are collectively termed clients.

There are two approaches to initiating event communication between suppliers and consumers referred to as the push model and the pull model. The push-model, which is considered the typical event communication model, allows the supplier to initiate the transfer of event data to consumers whereas the pull-model allows a consumer to request event data from a supplier. A consumer may use either a blocking (pull) or a non-blocking (try_pull) mechanism when polling for event data.

The CORBA Event Service allows suppliers and consumers to invoke each other's interface methods directly when exchanging event data. This approach requires clients to be aware of the object references of their peers and hence prevents anonymous event communication. Alternatively, clients may interact through an event channel acting as intermediate between suppliers and consumers. In order to connect to an event channel, clients need to obtain a reference to the channel.

The Event Channel Architecture

A CORBA event channel is an intermediate object that decouples communication between suppliers and consumers allowing them to interact anonymously. Figure 2.3 shows how consumers and suppliers connect to an event channel rather than directly to each other. An event channel acts as both a supplier and consumer of event data; it acts as a single consumer from a supplier's perspective and as a single supplier from a consumer's perspective. Any number of suppliers may issue event data to any number of consumers using a single event channel. There is no correlation between the number of suppliers and consumers connected to a channel and clients can be easily added to a channel.

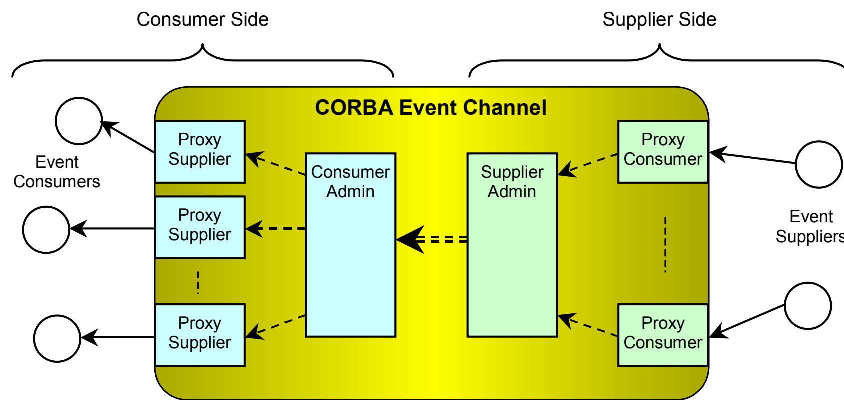


Figure 2.3. The CORBA event channel.

Consumers and suppliers may use the same or different communication models when interacting with an event channel. Figure 2.4 (A) and (B) show push-style and pull-style communication between supplier and channel and consumer and channel respectively. Figure 2.4 (C) illustrates an example of mixed style communication where a consumer uses the pull model to obtain event data from the channel while a supplier uses the push model to pass event data to the channel.

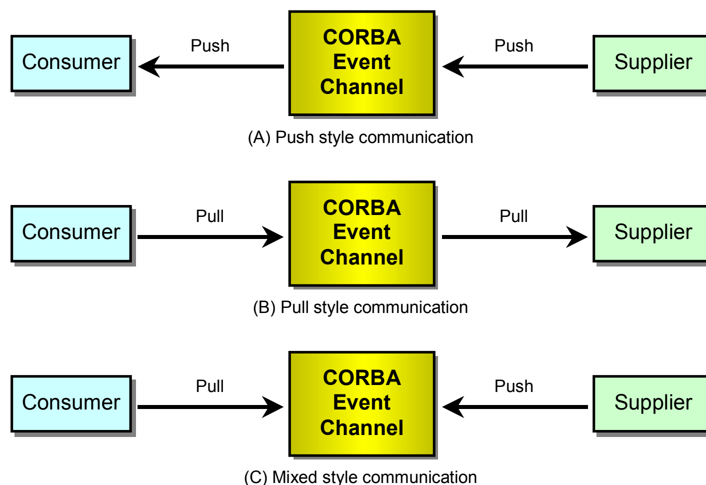


Figure 2.4. Communication styles.

Figure 2.3 also outlines the two sides of an event channel. The supplier side includes all interfaces used by suppliers and the consumer side includes all interfaces used by consumers. Clients connect to an event channel by obtaining an interface to a proxy object. Suppliers obtain proxy consumers each acting as single consumer and consumers obtain proxy suppliers each acting as single supplier. Clients obtain their proxies from the channel's

administration interfaces, called SupplierAdmin and ConsumerAdmin. These administration objects act as factory objects - objects that instantiate other objects. Essentially, clients register with a channel by obtaining proxy objects. Each client requires a separate proxy through which it exchanges event data with the channel. Event channels use administration objects for establishing and maintaining the connections to their clients and use those connections when propagating event data.

Figure 2.5 depicts a simplified version of some of the interfaces of an event channel which are described using the CORBA Interface Definition Language (IDL) [47]. The event channel interface defines two operations for clients to obtain the consumer and supplier administration interface. The ConsumerAdmin and the SupplierAdmin interface are similar. Both serve as object factories and each defines two operations for obtaining proxies. They define the operations for obtaining push and pull proxies for suppliers and consumers respectively.

```
Interface EventChannel{
   ConsumerAdmin for_consumers();
   SupplierAdmin for_suppliers();
}

Interface ConsumerAdmin{
   ProxyPushSupplier obtain_push_supplier();
   ProxyPullSupplier obtain_pull_supplier();
}

Interface SupplierAdmin{
   ProxyPushConsumer obtain_push_consumer();
   ProxyPullConsumer obtain_pull_consumer();
}
```

Figure 2.5. CORBA event channel interfaces.

Although the CORBA Event Service specification does not define additional event channel capabilities, a particular vendor may provide an event channel implementation supporting features such as event filtering and delivery semantics. An application may combine the features of different event channels by composing them. Event channel composition allows one channel to consume the event data supplied by another channel. However, tunnelling event data through multiple channels requires additional computational resources and increases event delivery latency.

In theory, any number of suppliers and consumers may connect to a specific event channel. As an event channel propagates all event data generated by its suppliers to all connected consumers (assuming event data is not filtered), the computational load of a specific channel increases with the number of its clients. The architecture of the CORBA Event Service

addresses this by implicitly supporting the use of multiple event channels within a system. In this approach, a client may connect to one or more channels, each propagating a subset of the event data in the system. This implies that a client needs to connect to the particular channel handling the event data in which the client is interested and as a result provides a means for implicit, subject-based filtering.

The CORBA Event Service specification does not define the means by which clients obtain references to event channels. It is therefore left to the application programmer to provide a mechanism such as a naming service for obtaining these references.

Generic and Typed Event Channels

An event channel may be implemented as either generic or typed. A generic event channel supports generic event data only whereas a typed event channel supports both typed and generic event data. Suppliers and consumers interacting through a generic event channel must agree on the content of their event data since generic event data are data blobs without an explicit structure. Typed event data is described using CORBA IDL. A typed event channel can handle event data supplied and consumed in any combination of the forms push/pull and generic/typed. Event data supplied in a typed form can be consumed in a generic form and vice versa.

The CORBA Event Service specification explicitly states that typed event channels support the conversion of typed into generic and generic into typed event data. However, it admits that this requires a profound understanding of the interfaces of an event channel and depends on the particular event channel implementation. Mapping typed into generic event data is relatively simple. However, converting a generic data blob into typed event data is challenging. [46] notes that many users have found typed event communication (described in IDL) difficult to understand and implementers have found it particularly difficult to deal with.

2.4.2 CORBA Notification Service

The CORBA Notification Service extends the CORBA Event Service by supporting typed events with a predefined structure, called structured events, filtering, and quality of service constraints.

The Notification Channel Architecture

The main design goal of the CORBA Notification Service was to directly extend the CORBA Event Service with the additional features listed above. This is achieved by deriving the

interfaces of the notification channel directly from those defined by the event channel allowing for interoperability between Event Service and Notification Service clients. As a result, the notification channel encapsulates all interfaces and functionality supported by the event channel.

Figure 2.6 outlines how the notification channel extends the event channel architecture by supporting multiple instances of both ConsumerAdmin and SupplierAdmin objects. Both the supplier and the consumer side of the notification channel allow clients to obtain proxy objects from any one of these administration objects. The symmetric nature of the notification channel and the administration objects are essential for the Notification Service capabilities described below.

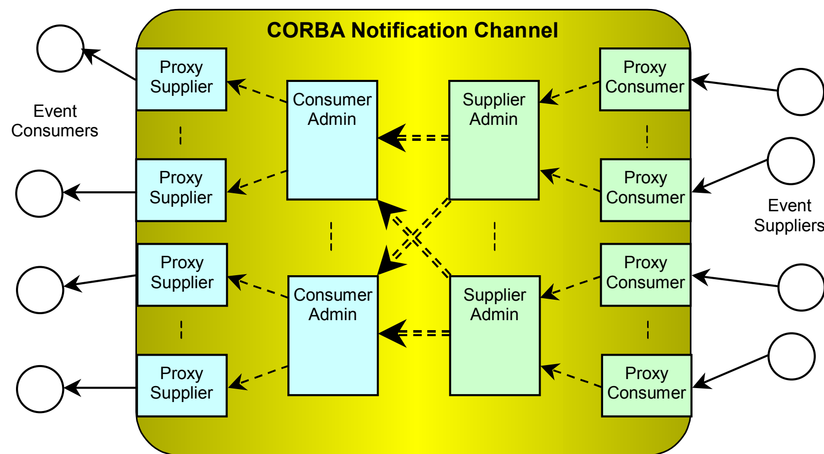


Figure 2.6. The CORBA notification channel.

The Notification Service specification defines an event type repository that may be used for discovering the structure of typed event data and for performing run-time type checking. However, using an event type repository is not essential for the Notification Service to operate correctly and consequently, using such a repository is optional.

Administrative Capabilities

Applications can configure various administrative properties when creating a notification channel in order to limit the use of memory space. As shown in Table 2.2, these properties include upper bounds on the number of suppliers and consumers that may be connected to the channel at any given time as well as on the number of events that may be stored. Notification channels may queue events if more events are supplied than can be

disseminated to consumers. Such a queue acts as buffer to prevent events from being discarded in case of a temporary surplus of supplied events.

Administrative Property	Possible Values
MaxSuppliers	0..max(long)
MaxConsumers	0..max(long)
MaxQueueLength	0..max(long)

Table 2.2. The administrative properties of notification channels.

As illustrated in Figure 2.6, a notification channel comprises a hierarchy of administration and proxy objects for maintaining client connections. Starting from one of these objects or indeed the channel object, clients may trace through the hierarchy discovering other objects. Factory objects assign a unique numeric identifier to every proxy or administration object they create and provide an operation for retrieving a list containing these identifiers. Notification channel objects support an operation returning the identifier of their parent object. These identifiers are unique among the objects created by a particular notification channel, but unlike object references [48], they are not globally unique.

The objects that comprise a notification channel provide operations for clients to specify the type of the event data they handle. Suppliers use the `offer_change` operation to indicate changes to the type of events generated and consumers invoke the `subscription_change` operation to inform suppliers of the event data type they require. Consequently, suppliers can know what event data is being consumed allowing them to suspend the generation of unwanted events in order to optimise network traffic.

Quality of Service

The Notification Service supports a variety of properties defining the QoS characteristics of the service that may be set to control the propagation of event data. Operations for setting these properties are specified on various objects throughout the Notification Service architecture including:

- The notification channel (per-channel)
- Supplier and consumer administration (per-admin)
- Proxy suppliers and consumers (per-proxy)
- Individual event messages (per-event)

The object on which a specific property is configured defines the scope to which the setting is relevant. The QoS configuration of a specific object applies to the object and to all its descendants. Table 2.3 summarises the QoS properties supported by the Notification Service and outlines the scopes to which they apply. Note that setting the MaxEventPerConsumer and DiscardPolicy properties on a per-SupplierAdmin or per-ProxyConsumer basis had no meaning. [12] discusses these QoS properties in detail.

QoS Property	Per-Event	Per-Proxy	Per-Admin	Per-Channel	Possible Values
EventReliability	✓			✓	BestEffort, Persistent
ConnectionReliability		✓	✓	✓	BestEffort, Persistent
Priority	✓	✓	✓	✓	-32767..32767
StartTime	✓				TimeBase::UtcT (absolute)
StopTime	✓				TimeBase::UtcT (absolute)
Timeout	✓	✓	✓	✓	TimeBase::TimeT (relative)
StartTimeSupported		✓	✓	✓	False, True
StopTimeSupported		✓	✓	✓	False, True
MaxEventPerConsumer		✓	✓	✓	0..max(long)
OrderPolicy		✓	✓	✓	AnyOrder, FifoOrder, PriorityOrder, DeadlineOrder
DiscardPolicy		✓	✓	✓	AnyOrder, FifoOrder, LifoOrder, PriorityOrder, DeadlineOrder, RejectNewEvents
MaximumBatchSize		✓	✓	✓	0..max(long)
PacingInterval		✓	✓	✓	TimeBase::UtcT

Table 2.3. Notification Service QoS properties.

The set of supported QoS properties combined with their scopes provide a flexible means for configuring the QoS characteristics of a notification channel. However, this approach requires the application programmer to prevent meaningless QoS settings. Event data passes through three conceptual points, namely supplier side, consumer side, and notification channel when being propagated from suppliers to consumers. All three of these parties have to cooperate in order to provide end-to-end QoS. For example, a Notification Service may be configured for assured event data delivery by setting persistent reliability (event and connection) and by assigning high priority and long lifetime to the event data. Such a configuration would enforce guaranteed event data delivery, but does not ensure predictable delivery latency.

Structured Event Data

The Notification Service introduces a kind of typed events called structured event data in order to provide an easy-to-use (compared to the Event Service's IDL-based approach) and strongly typed event communication mechanism. Structured event data provides a well-defined, expressive data structure into which a variety of event types may be mapped. As depicted in Figure 2.7, structured events consist of a header and a body.

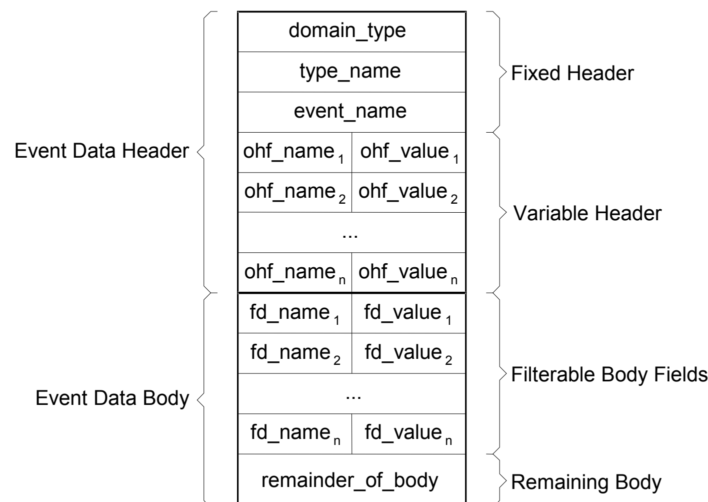


Figure 2.7. The general structure of structured event data.

The fixed part of the event header describes the type and instance identifier of the event data. The variable part may contain a number of attributes consisting of name-value pairs describing the QoS properties of the event data. The body of the event data comprises a variable number of attributes describing a specific event and a part that may be used for propagating large data blobs, such as files.

Filtering

The notification channel supports a hierarchical approach to filtering through the use of filters called filter objects. Filter objects may be assigned to individual proxy objects (proxy supplier and consumer), to admin objects (supplier and consumer admin), and to the notification channel. Filter objects that have been assigned to a channel apply to all admin and proxy objects as well as to all clients connected to these proxies. Similarly, filter objects associated with an admin object apply to all proxy objects and their clients, whereas filter objects assigned to a specific proxy object apply exclusively to the client. This approach allows filter objects to be applied symmetrically on both the supplier and the consumer side of the

notification channel. Hence, filtering may occur on the event data generated by a group of suppliers as well as on the event data propagated to a group of consumers. Applying filter objects close to the event data source generally reduces utilisation of system resources.

All filter objects relevant to a specific proxy are evaluated at the proxy regardless of the level in hierarchy to which they have been assigned. These filter objects are matched against the event data received by the proxy and events are forwarded if the matching produced a positive result. Filter objects encapsulate a set of filter constraints each consisting of a text string containing a boolean filter expression. The example of Figure 2.8 shows a filter object and how it refers to specific attributes of the fixed header and the body of structured event data. This filter matches events with certain domain type and type name combinations in conjunction with a body attribute with a particular name and value. Events matching these constraints will pass through the proxy, other events will be discarded.

```
(($domain_type == “Finance” and $type_name == “ExchangeRateUpdate”)
or
($domain_type == “Health” and $type_name == “PulseLow”))
and
(office == 7)
```

Figure 2.8. Notification Service event data filter object.

The syntax of the constraint expressions outlined in Figure 2.8 conforms to the constraint grammar defined by the Notification Service’s default filter constraint language, which is an extension of the CORBA Trader Constraint Language [49]. However, other, proprietary constraint languages may be used instead of this default language.

The Notification Service defines two types of filter objects. Forwarding event data filters, an example of which has been shown in Figure 2.8, affect the decision on whether to forward or to discard specific event data. Mapping event data filters influence the delivery policy applied to event data, they may change the characteristics of the event data delivery semantics defined in the variable part of the event data header. For example, a mapping event data filter might assign a different priority or set a new expiration time to an event.

2.4.3 Summary

Both the CORBA Event Service and the CORBA Notification Service specify an event model that defines the roles of event data supplier, event data consumer, and event or notification channel respectively. Events may either be propagated directly from suppliers to consumers

or may be tunnelled through a channel. Channels act as mediator between event suppliers and consumers allowing anonymous communication. Both services support a push-based and a pull-based model for event delivery. Events can be propagated in generic, typed, or structured form; structured events being supported by the Notification Service only. The Notification Service directly extends its predecessor the Event Service by providing advanced capabilities including filtering on structured events, QoS properties, and administrative features.

Both event models can be characterised as extremely general addressing the requirements of a variety of business domains including telecommunications, finance, and medicine and complex due to the large number of interfaces and properties. They allow consumers to implicitly subscribe and suppliers to implicitly announce their event data by obtaining the interface to a proxy object through which they connect to a channel.

Neither of the CORBA event models supports federated event channels. Federation enables a group of channels to connect together in a topology of arbitrary complexity and to cooperate when disseminating events among their clients in order to improve system scalability. The OMG has addressed this by issuing a Request For Proposal [50] for a management service providing the ability to configure, manage, and control a group of channels connected together in a topology of arbitrary complexity. A consortium submitted a proposed specification for such a service in December 1999, which subsequently resulted in the formal publication of the CORBA Management of Event Domains Specification [51].

2.5 OMG CORBA Event Model Extensions

This section presents two event services that extend the OMG CORBA Event Service by providing capabilities omitted by both the CORBA Event and Notification Service. The TAO Real-Time Event Service [31] supports QoS properties addressing the requirements of real-time applications. CONCHA [32] is based on a reliable multicast protocol and provides reliable and totally ordered event delivery semantics.

2.5.1 TAO Real-Time Event Service

The ACE ORB (TAO) [52] is an ORB-based real-time middleware developed at Washington University. TAO includes an extension to the CORBA Event Service called the Real-Time Event Service (RT Event Service) that addresses the requirements of distributed real-time applications. The RT Event Service has been designed for an avionics mission control

application and features real-time event dispatching, event filtering, and periodic event processing.

The RT Event Channel Architecture

The RT Event Service defines the roles of event supplier, event consumer, and event channel as in the standard CORBA Event Service. However, the event channel of the RT Event Service has been adapted to support additional features including *real-time* event dispatching and scheduling, source-based and type-based filtering, event correlation, and periodic event processing.

As depicted in Figure 2.9, the RT event channel consists of a set of main modules implementing these features. Each module may contain multiple “pluggable” strategies optimised for different requirements, allowing an application to select the one that appropriately addresses its needs. For example, the dispatching module may provide a range of pre-emption strategies from which an application may choose a suitable one. Moreover, some of these modules may be removed to optimise the RT event channel for certain configurations. For example, an application that has no complex inter-event data correlation dependencies may omit the correlation module.

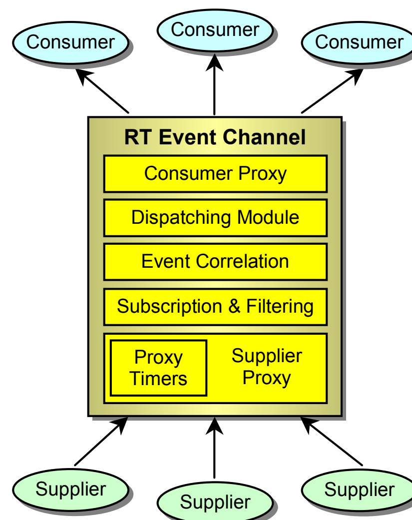


Figure 2.9. The TAO RT event channel.

Real-Time Event Dispatching

The RT event channel's standard proxy interfaces have been extended allowing suppliers and consumers to register their execution requirements with the event channel using QoS attributes. These attributes configure the dispatching mechanism determining event dispatching order and pre-emption strategy. The dispatching module implements priority-based event dispatching and pre-emption using priority queues. Harrison et al. [31] describe the pre-emption strategies supported including real-time upcall (RTU) dispatching, real-time pre-emptive thread dispatching, and single-threaded priority-based dispatching.

Filtering

The RT Event Service extends the event channel with a filtering mechanism that requires a well-defined type system for event data. The filtering module defines such a type system by including source identifier, type, and timestamp fields in event messages, thus allowing suppliers to describe the type of event data that they generate. Using these fields, an event channel provides supplier-based and type-based filtering. Supplier-based event data filtering allows consumers to register interest in events generated by certain suppliers whereas typed-based filtering lets consumers subscribe to event data of a particular type. Consumers may employ combinations of supplier-based and type-based filtering.

A further means to reduce the value of events propagated to consumers is provided by the event correlation module. The event correlation mechanism allows consumers to define dependencies between occurrences of certain events. Consumers may define conjunctive or disjunctive semantics when registering their event filtering requirements. Conjunctive semantics instructs a channel to notify consumers when *all* of the specified event dependencies are satisfied and disjunctive semantics compels a channel to notify consumers when *any* specified event dependencies are satisfied.

Periodic Data Event Processing

The supplier proxy module allows consumers to specify event data dependency timeouts that define time periods within which consumers expect to receive at least one event. Priority timers manage these timeouts and notify consumers by dispatching timeout events if their dependencies are not satisfied within this time period. This mechanism is well suited for periodic event data processing and for implementing real-time “watchdog” timers.

2.5.2 CONCHA

CONCHA (CONference system based on java and CORBA Event Service CHAnnels) provides extensions to the CORBA Event Service based on using the Light-weight Reliable Multicast Protocol (LRMP) [32] as the underlying transport mechanism. These extensions include reliable, multicast-based communication and totally ordered event delivery. LRMP is a reliable, general-purpose transport protocol based on unreliable underlying network transport protocols, such as UDP/IP. LRMP uses a sliding window buffer to support loss repair, ordered message delivery, flow control, and congestion control.

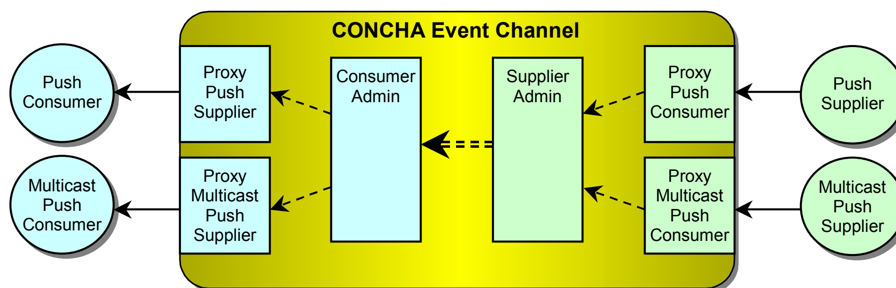


Figure 2.10. The CONCHA event channel.

Figure 2.10 outlines CONCHA's event channel architecture including the integrated multicast support. The CONCHA channel provides a single multicast proxy for each side of the channel. The proxy multicast push consumer handles all multicast suppliers and the proxy multicast push supplier deals with all multicast consumers. These proxies support the push-based communication model only and thus all multicast clients must be push based.

The CONCHA event channel allows clients to interact using either the standard event data propagation mechanism based on the Internet Inter-ORB Protocol (IIOP) [53] or the multicast extension providing reliable and ordered event data delivery. Event dissemination from suppliers to consumers is not limited to either the standard or the multicast mechanism. The two approaches may be combined allowing multicast suppliers to push event data to both multicast and standard consumers.

2.5.3 Summary

The TAO RT Event Service addresses the requirements of distributed real-time applications by extending the CORBA Event Service with efficient source and type-based filtering, event correlation, and, most importantly, with end-to-end real-time event dispatching. The

dispatching module uses priority queues to implement a variety of priority-based dispatching and pre-emption strategies. The RT Event Service uses a centralised event channel to implement these features and global application knowledge to statically configure a system based on a previously known number of clients. This approach does not scale well as such an event channel may become a communication bottleneck. However, it suffices in small-scale applications such as avionics mission control for which the RT Event Service has been designed.

CONCHA provides extensions to the CORBA Event Service based on using the LRMP as the underlying transport mechanism. Consequently, CONCHA supports reliable, multicast-based communication and totally ordered event data delivery. The CONCHA event channel implements a simple approach to multicast group management based on exploiting a single group through which all multicast event data is disseminated.

2.6 JAVA Event Models

Java is a general-purpose, concurrent object-oriented programming language developed by Sun Microsystems. The Java programming language is machine independent and supports strong typing as well as automatic storage management. Java has become increasingly popular because of Internet-related developments, such as the World Wide Web. In general, Java is valuable for building distributed, platform-independent applications.

Java source code is compiled into Java bytecode, which has been designed to run on a Java Virtual Machine (JVM). Bytecode is a language for an abstract machine and may execute on a virtual machine on any system that supports Java. Java supports two event models, known as the Delegation Event Model [24, 54] and the Distributed Event Model [13]. The Delegation Event Model is used for event communication within a single JVM and has been designed for small-scale, centralised applications, such as Graphical User Interfaces (GUIs). The Distributed Event Model enables event-based communication between objects in JVMs located in separate address spaces, possibly distributed across different physical machines.

2.6.1 Java AWT Delegation Event Model

The Abstract Window Toolkit (AWT) and its successor Swing, which are both part of the Java Foundation Classes library, are the standard application programming interfaces for providing graphical user interfaces for Java applications. The Java Foundation Classes initially supported an event processing model based on inheritance. However, this event model was

replaced by the Delegation Event Model with the introduction of version 1.1 of the Java Development Kit. Compared to the inheritance-based event model, the Delegation Event Model supports event filtering and a more robust framework for sustaining more complex Java applications.

The Delegation Event Model has been used by a number of other Java components. It has been adopted for general event processing by the JavaBeans [55] component architecture and for processing events on behalf of a new GUI toolkit, called the Swing Component Set [56]. Furthermore, both the EmbeddedJava [57] and the PersonalJava [58] application environment support event processing based on the Delegation Event Model. EmbeddedJava has been designed for building small-footprint applications that can be embedded in devices with dedicated functionality and strictly limited memory. PersonalJava is intended for building network-connected applications for consumer devices requiring near-desktop graphics capabilities for home, office, and mobile use, such as web phones, digital set-top boxes, personal digital assistants, and car navigation systems.

Architecture

Although this review focuses on distributed event models, we introduce the centralised Delegation Event Model since it is widely used as a result of its association with the Java programming language. The Delegation Event Model has been designed for small-scale GUI applications and is typically used for interconnecting GUI components residing in a single address space. GUI components acting as event sources propagate, or fire, events of a specific type to event listeners.

```
java.util.EventListener
   java.awt.event.ActionListener
   java.awt.event.TextListener

java.util.EventObject
   java.awt.event.ActionEvent
   java.awt.event.TextEvent
```

Figure 2.11. EventListener interface and EventObject class.

Figure 2.11 outlines parts of the EventListener interface and the EventObject class used by Java applications to implementing event-based communication. Event listeners must be derived from the EventListener interface and implement the event handler associated with the specific type of event in which they are interested. For example, in order to receive ActionEvents fired by buttons, event listeners implement the `actionPerformed` method of

the ActionListener class. The EventObject class defines various event types, including ActionEvent and TextEvent, encapsulating the semantics of user interface components. ActionEvents indicate that commands associated with objects, such as buttons and menu items, be executed whereas TextEvents describe a change to the value of a text object.

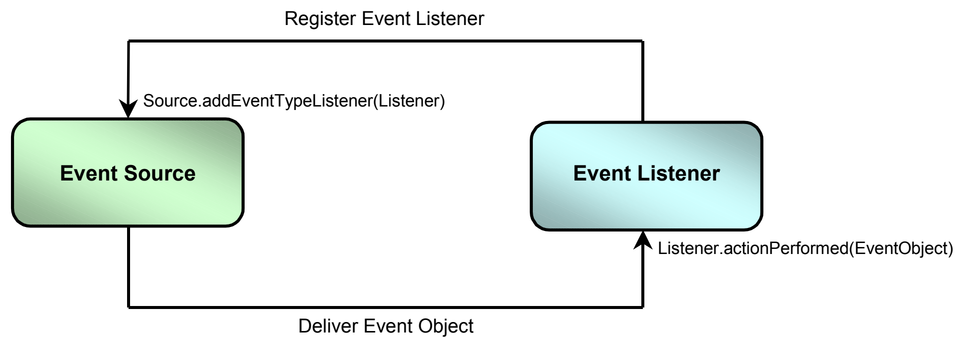


Figure 2.12. The Java Delegation Event Model.

Figure 2.12 illustrates that listeners subscribe directly at a particular source by invoking either the `set<EventType>Listener` or the `add<EventType>Listener` registration method passing a reference to their event handler. For example, a listener may subscribe to the ActionEvents fired by a specific button by invoking the button's addActionListener method. Every source provides both registration methods for each supported event type. The `set<EventType>Listener` method registers a single listener whereas the `add<EventType>Listener` method allows multiple listeners to subscribe to the same event type. This approach delivers events synchronously as the listener's handler is actually executed by the source thread and even multicast sources deliver specific events to their listeners sequentially. However, the Delegation Event Model provides no guarantees regarding the order in which a particular event will be delivered to a group of listeners. Significantly, direct registration results in a tight and explicit coupling between an event source and its listeners that does not support anonymous communication.

Event Adapter

As shown in Figure 2.13, a Java application may employ a component, called an event adapter, between event source and event listener to partially decouple their communication. Event adapters allow applications to introduce additional behaviour on event delivery, such as event queuing and filtering. They register with a source on behalf of one or more listeners and subsequently forward events to these listeners.

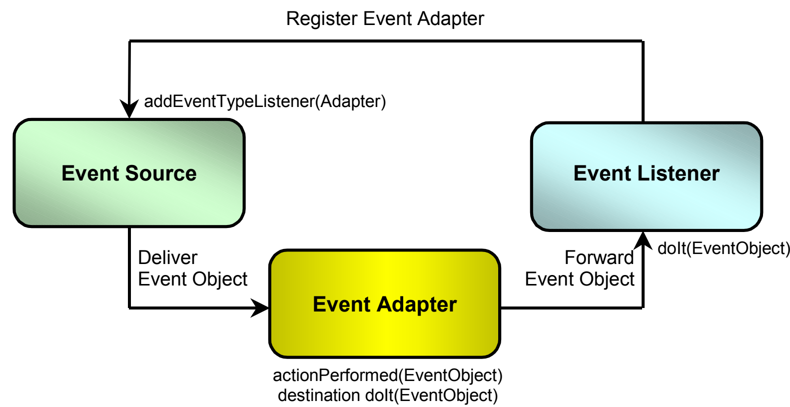


Figure 2.13. Using event adapters in the Delegation Event Model.

In contrast to other event models supporting intermediate components, such as CORBA Event and Notification Services, event adapters are asymmetric in that they hide event listeners from sources, but not vice versa. They introduce a notion of anonymity where listeners are anonymous although sources and adapters are not.

2.6.2 Java Distributed Event Model

The Java Distributed Event Model allows an object located in one JVM to receive events from objects in another JVM and relies therefore on Remote Method Invocation (RMI). The Distributed Event Model has been adopted by Jini [59], a Java technology that provides a simple mechanism for enabling the spontaneous assembly and interaction of services and devices on a network. Jini is typically used for plugging together network devices forming a communication community without any planning or installation.

Architecture

The Distributed Event Model is similar to the Java AWT Delegation Event Model in that it specifies the interface used to deliver events, defines the information that an event must contain, and explicitly supports interposing objects. However, it omits specifying a registration interface in order to allow a wide variety of kinds of events. The kind of an event is defined by the specific object in which the event occurs. Consequently, the way in which interest in such events is registered depends on the particular application and may vary from object to object.

```
public interface RemoteEventListener extends Remote,
java.util.EventListener {
   void notify(RemoteEvent theEvent)
           throws UnknownEventException, RemoteException;
}

public class RemoteEvent extends java.util.EventObject {
   public RemoteEvent(Object           source,
                      long             eventID,
                      long             seqNum,
                      MarshalledObject handback)
}
```

Figure 2.14. RemoteEventListener interface and RemoteEvent class.

The Distributed Event Model defines the role of event generators supplying remote event objects and passing them to registered remote event listeners. Figure 2.14 outlines the `notify` operation for synchronously delivering events to remote event listeners and shows that remote events contain information describing the event that occurred, including a reference to the generator, a sequence number, and a handback supplied by the listener. This sequence number identifies a specific event instance and is guaranteed to be strictly increasing. The Jini specification [59] states that the sequence number is relative to a previous sequence number and that the sequence number of two event objects differs if and only if the event objects are a response to different event occurrences. However, it is not clear whether a sequence number is relative to an event generator or to each registered event listener and as a result, what initialises a sequence.

The Distributed Event Model allows remote event listeners to limit the duration of their subscriptions using the concept of leasing, which has been defined by the Java Distributed Leasing Specification [60]. Such subscriptions expire after a certain leasing period and subsequently deregister listeners automatically.

Distributed Event Adapter

As shown in Figure 2.15, the Distributed Event Model supports interposing third party objects, called distributed event adapters, similar to event adapters in the Delegation Event Model. Distributed event adapters allow applications to specify additional functional and non-functional properties without changing the basic interfaces of the event model as long as such adapters support the `notify` method. They may provide various degrees of delivery guarantees, different event delivery policies, and may act as event filter or mailbox collecting, storing, filtering, and forwarding event objects on behalf of a single or a group of listeners.

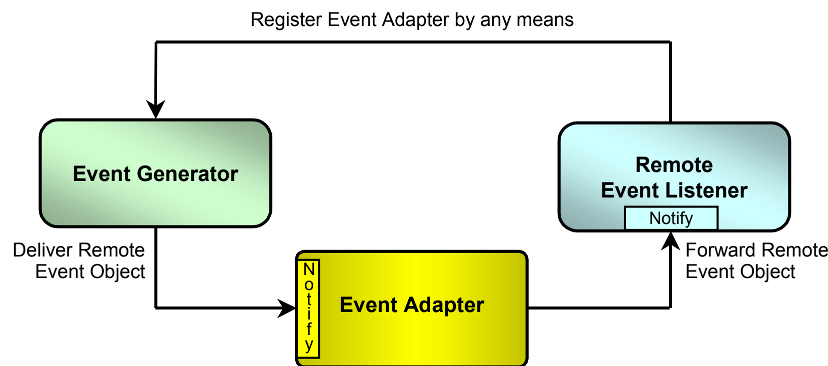


Figure 2.15. Using distributed event adapters in the Distributed Event Model.

The physical location of event adapters is of no importance from a conceptual perspective. They may be collocated with either event generators or listeners sharing the same address space, or they may reside in a separate address space possibly on a designated physical machine. Numerous event adapters can co-exist in a system and may co-operate in order to combine their properties.

As described above, using event adapters introduces a notion of anonymity in which listeners are anonymous whereas sources and adapters are not. However, using adapters increases the complexity of a system due to additional communication and computational overhead and consequently increases event delivery latency.

2.6.3 Summary

The Java programming architecture supports a Delegation Event Model for processing events in small-scale, centralised applications and a Distributed Event Model for event-based communication between objects residing in different JVM's. The Distributed Event Model uses Java RMI as the underlying mechanism for inter-process communication. Both the Delegation and the Distributed Event Model have been adopted by a number of Java components and environments, including AWT and Jini respectively.

Both event models have a similar architecture in which generators propagate event objects to listeners while supporting interposing event adapters. Adapters allow applications to enhance a system with specific functional and non-functional properties without changing the basic interfaces. These adapters may be used to provide an application specific means for filtering events thereby supporting a feature lacked by both event model's.

Both event models require event listeners to register directly at the objects generating events of interest. However, the Distributed Event Model does not specify a specific method for doing so. It has been designed for a wide variety of applications using different kinds of events and as a result, the means by which interest in such events is registered may vary from application to application. The Distributed Event Model supports the concept of leasing for automatically deregistering event listeners after a certain leasing period has elapsed.

2.7 CEA

The Cambridge Event Architecture (CEA) [4, 33], which has been developed by the Opera Research Group of the Computer Laboratory at the University of Cambridge, is based on an event model that supports composite events and producer-side filters, called parameter templates.

2.7.1 Cambridge Event Architecture

CEA has been designed to extend widely used middleware platforms, such as CORBA, Java RMI, and DCOM [61], allowing producers and consumers, called source objects and client objects, to interact using event-based operations. Event source objects specify the type of their events as well as their registration interface in IDL and then publish these interfaces using a means provided by the underlying middleware platform. Client objects use these registration interfaces to subscribe to events. Ma and Bacon [62] describe how CEA has been added to a CORBA implementation.

```
Badge : INTERFACE = Seen : EVENTCLASS [badge  : BadgeId;
                                       sensor : SensorId];
END.

e = Badge_Seen(17, 29);
EventSource.Signal(e);
```

Figure 2.16. An event source object of an active badge system.

Bacon et al. [33] describe an example of an active badge system in which source objects use CEA to signal sightings of badges by certain sensors to client objects. Figure 2.16 and Figure 2.17 show a simplified version of this example. Figure 2.16 outlines a source object specifying an event type, called “Badge_Seen”, comprising two numerical values describing the person and location of a specific sighting and subsequently raising an event of this type.

Figure 2.17 illustrates a client object specifying and then registering an event parameter template that matches a specific badge seen by a certain sensor.

```
Template = Badge_Seen(17, 29);

EventClient.Register(EventHandler, template);
```

Figure 2.17. An event client object of an active badge system.

Figure 2.18 shows a high level overview of CEA, illustrating the two approaches for source objects to asynchronously disseminate events to client objects. As shown in Figure 2.18 (A), source objects may notify client objects directly of events. In this scenario, source objects match their events against registered templates prior to publication. Generally, source-side filtering minimises the use of communication resources as only wanted events are transmitted while distributing the computational load for template evaluation. Figure 2.18 (B) depicts a mediated CEA, in which the filtering function is removed from potentially primitive source objects. Instead, events are disseminated through a mediator, which may be used to match templates as well as more complex filter expressions, such as event composition operators. Moreover, mediators may support mobile applications, buffering and forwarding of events for temporarily disconnected nomadic client objects.

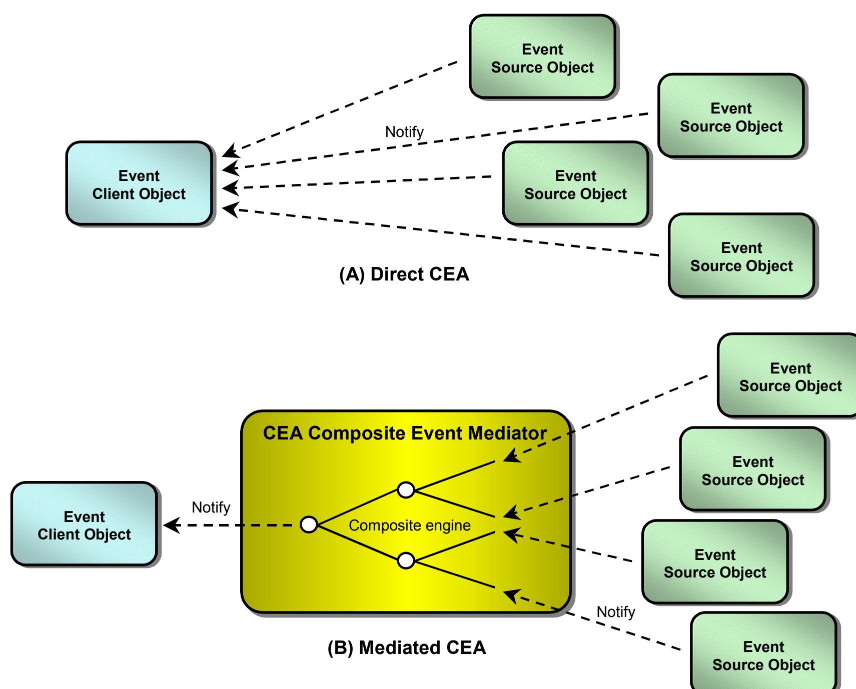


Figure 2.18. Overview of the Cambridge Event Architecture.

Filtering

CEA allows client objects to specify event filters known as parameter templates. As shown in Figure 2.19, such a template consists of the name of an event type and a set of ordered arguments, each specifying either a specific event parameter value or a wild card variable matching any event parameter value. A template matches if its name and its arguments match. These arguments are matched one-by-one against the parameters of a specific event instance, i.e., the first template argument will be matched against the first parameter of an event, the second template argument will be matched against the second event parameter, and so on. In general, the approach of using templates for defining event filters can be characterised as simple but of limited expressiveness since all operators are implicitly predefined. Template expressions imply an equality operator when comparing arguments with event parameters and assume a conjunctive relationship between argument-parameter pairs.

```
Template = EventTypeName(arg1, arg2, .., argn);

templateWhere = Badge_Seen(17, Room);
templateWho   = Badge_Seen(Person, 29);
templateGod   = Badge_Seen(Person, Room);
```

Figure 2.19. Defining event filters using parameter templates.

Figure 2.19 also outlines a number of template examples that may be applied to the event type defined in Figure 2.16. The “where” template includes a wild card variable that allows a client object to track a person wearing a certain badge in any room, thus providing information on the whereabouts of this person. The “who” template matches all sighting events raised at a certain location. The “God” template may be used by a surveillance system recording the movements of all people anywhere in the building.

Composite Events

Some applications may require their client objects to subscribe to events disseminated by multiple source objects and to detect a specific pattern of occurrences of these events. Moreover, such client objects may only be interested in the combination of event occurrences but not in any individual event alone. CEA addresses this requirement by allowing client objects to specify composite event filters and, as depicted in Figure 2.18(B), to use a mediator acting as composite event server to perform filtering across events of different types from various sources.

```
EventTypeNameA(arg1, .., argn); EventTypeNameB(arg1, .., argn); ..
FireAlarm(7); Badge_Seen(7, Person, Room);
```

Figure 2.20. Defining composite event filters using multiple parameter templates.

Bacon et al. [33] present a technique for easily specifying composite event filters and discuss how to register such templates with a composite event server that runs a monitor checking for sequences of events matching composite filters. Essentially, composite event filters are defined by combining multiple event parameter templates. Figure 2.20 shows the generic form for defining composite event filters as well as an example extending the previously discussed surveillance scenario based on an active batch system. This example monitors a specific building for the location of any person residing in the building after the filter alarm has been triggered.

2.7.2 Summary

CEA supports a composite event model and source-side event filtering based on parameter templates. It has been designed to extend commercially available middleware platforms, such as CORBA and DCOM, allowing source and client objects to specify their interfaces using an IDL. A pre-processor is subsequently used to translate the IDL code into source and client stubs for marshalling and un-marshalling of method invocations. Event source objects may disseminate events either directly to subscribed client objects or through an intermediate component. Such a mediator may match filter templates on behalf of its source objects and can act as a monitor to detect composite event occurrences.

CEA's approach of using parameter templates for event filtering is limited in its expressiveness since the operators for comparing event parameter values and template arguments as well as for combining parameter-argument pairs are predefined. However, this may be compensated by exploiting composite event filters, which allow client objects to subscribe to complex sequences of event occurrences. Significantly, CEA omits defining a framework for QoS enforcement and consequently it is left to a particular implementation to address the quality requirements of their application area by exploiting the capabilities provided by the underlying middleware.

Recent work that has been inspired by CEA includes Hermes [63], a distributed event-based middleware that proposes a scalable routing algorithm, and a general composite event

detection framework for distributed applications [64]. The latter formalises both time and event models of the proposed composite event detector.

2.8 ECO

The ECO event model was initially designed to interconnect components in the VOID shell [65]. The VOID shell is a system for distributed virtual world support, which was developed at Trinity College Dublin as a part of the Moonlight [65] project. A version of the ECO event model, called ECOlib [66], was implemented as a central part of VOID. ECOlib supports event filters, called notify constraints, pre-constraints, and post-constraints.

Other extensions to the ECO model were implemented by O'Connell et al. [34], called DECO (Distributed ECO), and by Haahr [67], named SECO (Scalable ECO). DECO supports precompiled, statically linked notify constraints and provides a scoping mechanism based on the notion of zones. DECO relies on an underlying group communication mechanism as the means for components to interact. SECO features dynamically linkable notify constraints and uses synchronous inter-component communication. Haahr et al. [14] describe the uSECO version of SECO, which is based on unicast communication, and the mSECO version which uses a multicast communication pattern.

2.8.1 ECO Architecture

Starovic et al. [68] describe Events, Constraints, and Objects (ECO), which represent the three principal concepts used in the ECO event model. Objects are instances of classes that have attributes and methods. These objects represent encapsulated application components that cannot directly access each other’s attributes or invoke each other’s methods, but may interact by disseminating and processing events of certain types. ECO supports various types of constraint in order to allow applications to specify their event propagation and synchronisation requirements. These constraints include notify constraints acting as producer-side event filters as well as pre-constraints and post-constraints, which may be used for implementing synchronisation and concurrency requirements. The ECO event model does not specify how constraints be defined or implemented and consequently, it is left to a particular implementation to support constrains that adequately address the requirements of the envisaged application domain.

```
Subscribe(eventType, eventHandler, notifyConstraint,
          preConstraint, postConstraint)

Unsubscribe(eventType, eventHandler)

Announce(eventType, eventParameters)
```

Figure 2.21. The application programming interface defined by ECO.

Figure 2.21 outlines the generic Application Programming Interfaces (APIs) defined by the ECO event model for subscribing, unsubscribing, and announcing events. The `announce` method is invoked by producers in order to disseminate events and should not be confused with the concept of announcing event types, which has been introduced in section 2.2.3.

Filtering using Pre-Constraints and Post-Constraints

Pre-constraints and post-constraints act as method wrappers for event delivery handlers and represent a powerful means for applications to control event delivery. An object processing events evaluates its pre-constraints prior to invoking a handler and applies post-constraints after delivering an event. Hence, pre-constraints and post-constraints may be used to enforce a wide range of event delivery properties. For example, they may be used to synchronise the delivery of multiple events and may control the concurrency level within a delivery handler or an event processor. Furthermore, they may implement timing controls, such as start time for earliest delivery and end time for latest delivery. In addition, pre-constraints may implement policies for queuing, discarding, or processing of events prior to their delivery.

Zones

The ECO event model allows applications to bound the scope within which events are disseminated by supporting the concept of zones. Objects associated with a particular zone are said to be members of this zone. Events may be propagated to members of a particular zone and will be delivered subject to matching constraints. Zone membership is determined according to functional or geographical aspects and the number of member objects may change dynamically. Moreover, a particular object may simultaneously become a member of one or more zones.

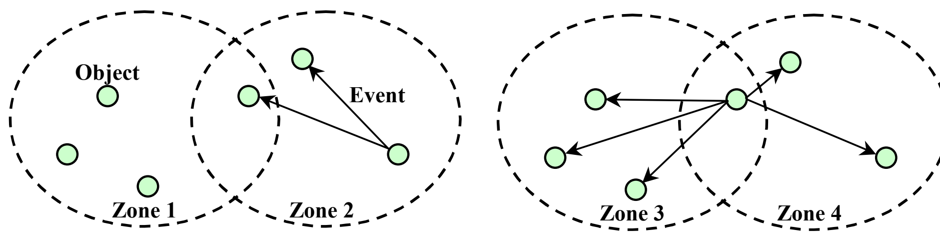


Figure 2.22. Overlapping zones.

An example scenario using zones might include a robot in a smart building subscribing to events describing the status of doors and to alarm events. Such a robot might join a geographical zone comprising the doors on the floor where it resides and a functional zone disseminating alarm events generated by sensors anywhere in the building. This scenario also demonstrates the dynamic membership [69] aspect of zones as the robot has to change its membership when moving from one floor to another. Moreover, a robot changing its membership might have to create a new zone before joining and might have to delete the old zone after leaving.

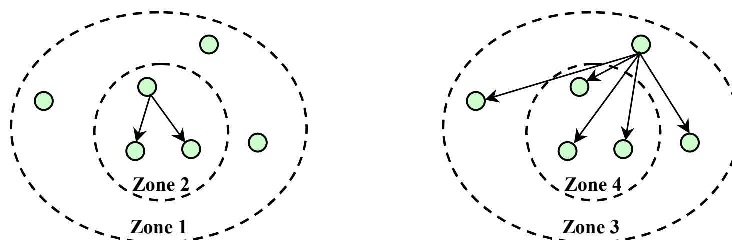


Figure 2.23. Nested zones.

O'Connell [70] describes the variety of zones defined by ECO. Figure 2.22 illustrates the concept of *overlapping zones* in which an object may become a member of several zones simultaneously. Figure 2.23 shows *nested zones* allowing zones comprising many objects to be subdivided. Figure 2.24 depicts a scenario in which an object that is not a member of a certain zone disseminates events in that zone; such an object is said to *target a zone*. Generally, these notions of zones may be combined to from other concept of zones. For example, overlapping and nested zones may be integrated to form a group of zones in which overlapping zones contain nested zones.

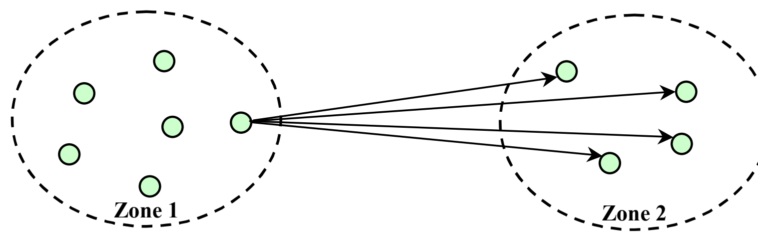


Figure 2.24. Targeting zones.

2.8.2 Summary

The ECO event model specifies the three basic operations required when providing event-based communication, namely subscribe, unsubscribe, and announce. In addition to supporting notify constraints, ECO introduces the concept of pre-constraints and post-constraints, which act as wrappers for an event delivery handler, and the notion of zones bounding the scope of event propagation. ECO implies a group-based approach for inter-object communication and consequently, does not rely on designated mediator components for event dissemination.

Instead of using a mediator component for enforcing delivery semantics, ECO allows applications to use pre-constraints and post-constraints to implement subscription specific policies for synchronisation, concurrency, timing control, queuing, and discarding. This approach provides a very flexible means for controlling event delivery. However, depending on the application domain, such a degree of flexibility may be inefficient as every object processing events might define similar pre-constraints and post-constraints. Moreover, choosing a delivery strategy based on object-local requirements may interfere with other system properties, such as event ordering and delivery guarantees.

ECO proposes several different concepts of potentially overlapping functional and geographical zones. Significantly, zones bound the scope of event dissemination and as a result limit the use of communication and computational resources. In addition, the concept of subdividing a system into bounded scopes may be particularly useful for managing large-scale systems.

2.9 JEDI

The Java Event-based Distributed Infrastructure (JEDI) [5] is an object-oriented infrastructure for the development of event-based applications that has been implemented in the Java

programming language. JEDI has been developed at Politecnico di Milano and has been used to implement a workflow management system, called the ORCHESTRA Process Support System (OPSS) [5], as well as the PROSYT process support system [71]. PROSYT supports applications comprising mobile agents migrating from host to host in local or wide area networks.

2.9.1 JEDI Architecture

JEDI introduces the notion of Active Object (AO) and Event Dispatcher (ED). Active objects are autonomous entities acting as either producers or consumers of events. They interact by disseminating events through an intermediate event dispatcher. Event dispatchers support `subscribe` and `unsubscribe` operations allowing active objects to register (or cancel) their interest in events and are also responsible for delivering events.

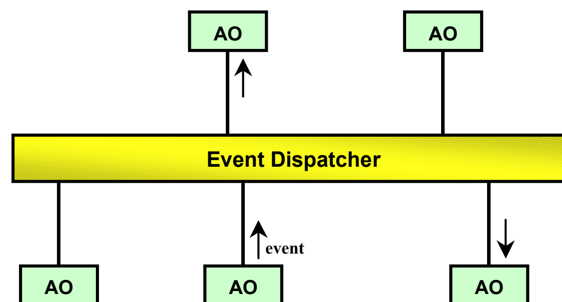


Figure 2.25. Overview of the JEDI architecture.

Figure 2.25 shows an overview of the JEDI architecture and outlines the logically centralised nature of the event dispatcher. An event dispatcher maintains global knowledge of all events and subscriptions issued in a system and consequently might become a performance bottleneck with increasing system scale. JEDI addresses this by providing a centralised and a distributed version of the event dispatcher. The centralised version comprises a single event dispatching component that has been designed to address the requirements of simple, small-scale applications disseminating a limited number of events. In contrast, the distributed version consists of a set of interconnected Dispatching Servers (DS) designed for large-scale applications comprising numerous active objects distributed over an Internet-scale network.

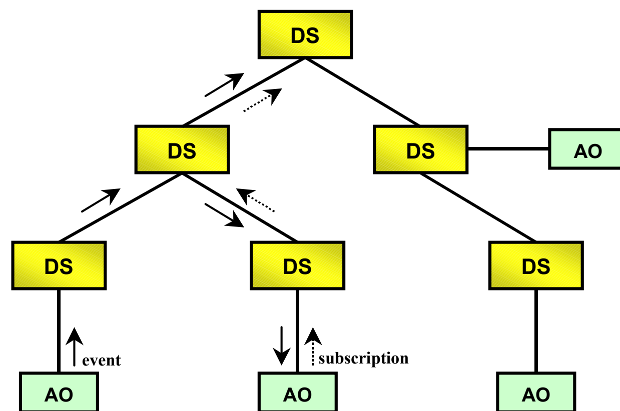


Figure 2.26. JEDI dispatching server topology.

Figure 2.26 outlines the topology of JEDI dispatching servers interconnected in a tree structure. Each dispatching server resides on a different physical node and connects to a parent server (unless it is the root server) and to zero or more child servers. Active objects may connect to a distributed event dispatcher through any dispatching server.

Events and subscriptions are distributed among dispatching servers according to a hierarchical routing strategy, thereby ensuring that all nodes receive relevant events and subscriptions. Starting from the issuing active object, subscriptions are passed upwards until they reach the root server causing the servers along the path to update their routing tables. Events are passed up the tree until they reach the root, while each server along the path passes them to any descendant with a subscription in its routing table.

Events

JEDI defines events as a set of ordered strings. The first string represents the name of an event and the remaining strings represent event parameters. Hence, events can be defined using a notation similar to function calls in traditional programming. Figure 2.27 shows examples of JEDI events that might be used to send a print job to a certain printer and to deliver meeting minutes to the members of a research team. Print and memo are the names of the respective events, both of which contain a number of parameters.

```
print(MyDocument, OurLaserPrinter)

memo(MeetingMinutes, John, Mary, Peter, Paul, Susan)
```

Figure 2.27. JEDI events.

Filtering

JEDI supports a simple form of event filters known as event patterns. Active objects may register interest in either a specific event or an event pattern. Event patterns are defined similarly to events as a set of ordered strings. However, each string of the pattern may end with an asterisk. In essence, an event matches a pattern if they have the same number of parameters and their name and parameters match, with asterisks representing any character sequence.

Reactive Objects and Mobility

JEDI supports mobility through the use of mobile agents called reactive objects. Reactive objects are a particular type of active object; they can be serialised using standard Java facilities and have the ability to autonomously migrate across the nodes of a network. JEDI provides the `moveOut` and `moveIn` operations allowing a reactive object to temporarily disconnect from the event dispatcher. The `moveOut` operation disconnects a reactive object from its current dispatching server, allowing it move to another location and then to use the `moveIn` operation to reconnect to another dispatching server at a later time. The event dispatcher stores subscription information on behalf of a moving reactive object preventing it from having to re-subscribe when reconnecting. Furthermore, the event dispatcher buffers relevant events while a reactive object is disconnected and delivers them upon reconnection.

2.9.2 Summary

JEDI provides a simple, easy to understand architecture for event-based communication. Its architecture is based on two components, namely active objects and event dispatchers. JEDI supplies both a centralised and a distributed version of the event dispatcher. The distributed version, which has been designed for Internet-scale applications, comprises a set of dispatching servers interconnected in a hierarchical structure. The hierarchical topology of the distributed dispatcher improves overall robustness and scalability of a system. However, disseminating subscriptions and events through multiple dispatching servers and the consequential inter-server cooperation activities require additional communication and computational resources. The resulting overhead might be significant for servers located close to the root of the hierarchy as they handle the most network traffic. Moreover, every server represents a critical point of failure for the whole network, since a failed server results in network segmentation.

JEDI allows applications to define events that comprise a set of ordered strings representing event name and parameters. Active objects may register interest in either a specific event or an event pattern. Event pattern provide a simple means for event filtering in which pattern strings that may contain asterisks are matched to event strings.

JEDI supports nomadic applications through a particular type of active object, called a reactive object. Reactive objects may temporarily disconnect from an event dispatcher while moving to their destination where they re-connect. The event dispatcher handles disconnection by buffering subscription and event information on behalf of a moving object and by forwarding relevant events when a reactive object reconnects.

2.10 SIENA

The Scalable Internet Event Notification Architecture (SIENA) [15] was also developed at Politecnico di Milano and is based on an architecture similar to JEDI. SIENA has been designed to support event based communication in wide-area networks, such as the Internet, and extends the JEDI architecture with a number of additional features. SIENA supports events that can have a more expressive structure than JEDI events, event filters that can be applied to such structures, and a range of event server topologies.

2.10.1 SIENA Architecture

SIENA has been designed to provide a scalable, general-purpose event model. As shown in Figure 2.28, SIENA supports event producing Objects of Interest (OIs) and event consuming Interested Parties (IPs). Event clients interact through a distributed set of interconnected event servers, which act as access points for clients and as store-and-forward network routers. These event servers cooperate with each other in order to provide a network-wide event service.

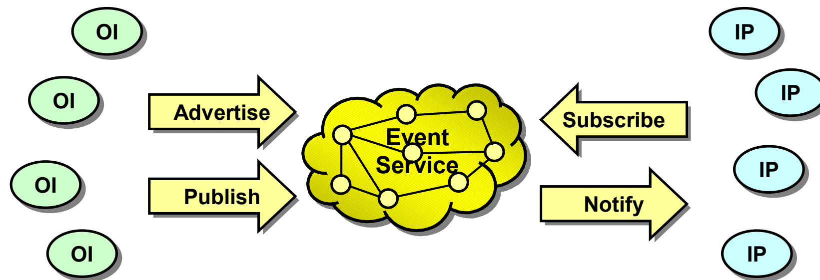


Figure 2.28. Overview of the SIENA architecture.

SIENA supports an advertisement and a subscription mechanism to regulate the propagation of events. These mechanisms provide a means for applications to describe the relationship between individual objects of interest and interested parties. This information can then be used to identify the best routes for disseminating events from objects of interest to interested parties thereby optimising the network traffic between event servers.

```
Advertise(objectOfInterest, filter)
Unadvertise(objectOfInterest, filter)

Subscribe(interestedParty, pattern)
Unsubscribe(interestedParty, pattern)

Publish(event)
```

Figure 2.29. SIENA application programming interface.

Figure 2.29 outlines the application programming interface supported by event servers, including the operations for advertising and subscribing. SIENA uses a string-based naming scheme that requires event client identifiers used for subscribing and advertising be unique throughout a system. This enables the service to maintain a mapping between the identities of interested parties and their event delivery handlers. Notably, both subscribers and advertisers may define filters passing them to the service during registration. Objects of interest advertise their intention to publish events that match their filters while interested parties define filters, called patterns, that describe their events of interest.

Operational Semantics

SIENA supports two alternative semantics for event delivery, namely subscription-based and advertisement-based. In the subscription-based version, event delivery is solely determined by subscriptions and matching patterns as advertised filters are ignored. The advertisement-based version enforces both advertisements and subscriptions in order to determine event

delivery. A particular event is delivered if the event service has received an advertisement and a subscription with a matching filter and pattern respectively.

Server Topologies

As summarised in Figure 2.30, SIENA supports four different topologies for interconnecting its event servers. Both the centralised client/server topology and the hierarchical client/server topology are similar to the architectures supported by JEDI. They use a client/server protocol for event clients to interact with an event server. The hierarchical client/server topology uses the same protocol for server to server communication. Consequently, an event server does not distinguish between descending servers and clients. The acyclic peer-to-peer topology and the general peer-to-peer topology have been introduced to address the shortcomings of centralised and hierarchical architectures discussed in section 2.9. Acyclic topologies comprise exactly one route between any two event servers whereas general topologies characteristically contain multiple routes between servers. Event servers in these architectures communicate with each other as peers using a server/server protocol, thus allowing bi-directional information flow. As SIENA has been designed for the Internet, such server/server interaction is typically based on a standard network protocol, such as the HyperText Transfer Protocol (HTTP) or the Simple Mail Transfer Protocol (SMTP).

In practice, these topologies may be combined to form hybrid server architectures. For example, the root servers of a group of hierarchical subnetworks might be interconnected through a general peer-to-peer topology forming a wide-area server architecture.

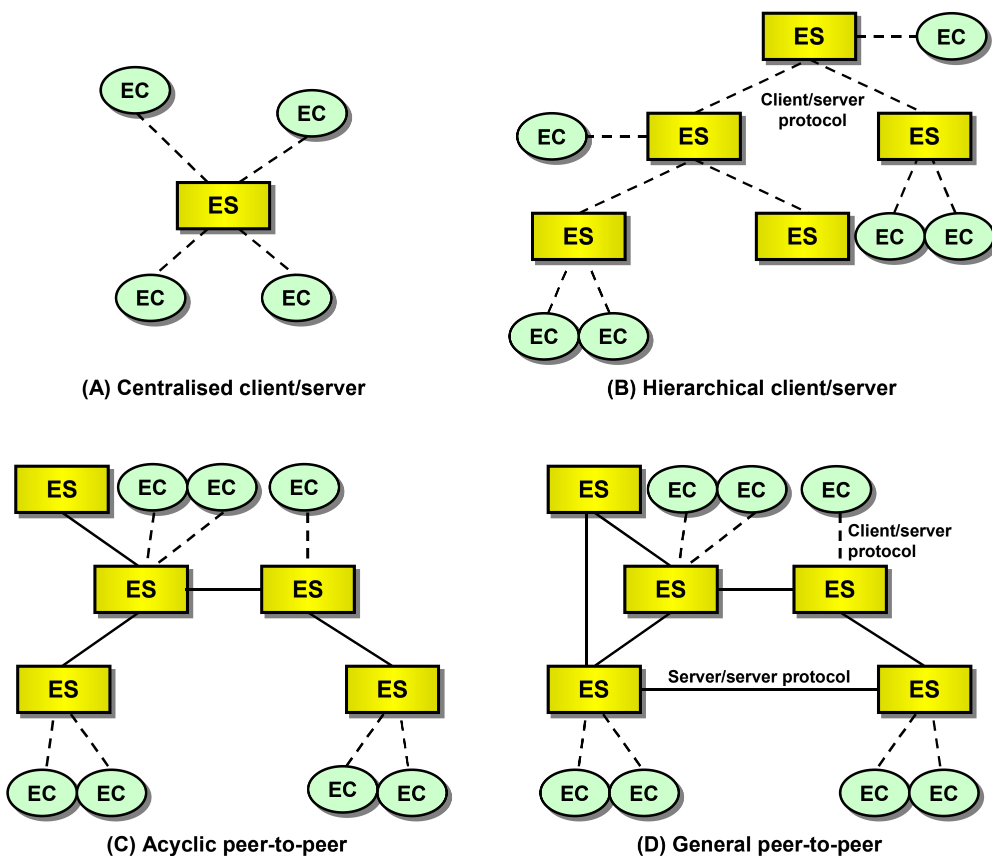


Figure 2.30. SIENA event server topologies.

Carzaniga et al. [15] formulate two generic principles that have been employed for optimising SIENA's routing strategies. The principle of downstream replication causes events destined for multiple interested parties to be replicated as close as possible to these parties whereas the principle of upstream evaluation states that event filters should be applied as close as possible to objects of interest.

Events

SIENA supports events in the form of a set of attributes in which each attribute is a triple of type, name, and value. Filter and pattern expressions may be matched against one or more of these attributes. The attribute types belong to a predefined set of primitive types commonly supported by programming languages. The example of Figure 2.31 shows an event indicating the time and location at which a door has been opened.

```
String event    = door/open
time   time     = 10:37:15
float  floor    = 2
float  building = 15
```

Figure 2.31. SIENA event.

Filtering

An event filter is defined by a set of attribute names and types combined with some constraints on the values of these attributes. Each constraint consists of a value and an operator. A fixed set of operators is available to define event filters of the form shown in Figure 2.32. The example filter might be used by a security system for monitoring door activities in a certain building prior to business hours.

```
String event    == door/*
time   time     <= 08:30:00
float  building == 15
```

Figure 2.32. SIENA event filter.

Event patterns define a combination of multiple event filters allowing applications to subscribe to correlated event occurrences. SIENA supports event patterns that combine filters according to an "A followed by B" semantics. Such patterns match if an event matching filter A precedes an event matching filter B. For example, Figure 2.33 shows a pattern that monitors a building for door activities at night combined with sightings of a person wearing a certain identification card.

```
String event    == door/open
time   time     >= 20:30:00
float  building == 15

followed by

float  card_id  == 1234
float  building == 15
```

Figure 2.33. SIENA event pattern.

Mobility

The version of SIENA described by Carzaniga et al. [15] does not support mobility. However, they state that they plan to enhance the SIENA architecture to support mobile event clients. Caporuscio et al. [72] propose to support nomadic clients by implementing a mobility support service for transparently managing subscriptions and events on behalf of moving clients. This mobility support service provides client proxy components that run on event servers acting as access points for clients and buffer events while disconnected clients move from one access point to another. Mobile clients connect to and disconnect from client proxies using an interface supporting `moveIn` and `moveOut` operations similar to the operations defined by JEDI. Caporuscio et al. [73] evaluate the performance of SIENA in a wireless network based on GPRS technology where event clients hosted by nomadic devices interact with event servers through wireless connections. These servers are interconnected through a backbone of a fixed wide-area network, such as the Internet.

2.10.2 Summary

The SIENA infrastructure implements a general-purpose event service that is based on an architecture of distributed event servers through which clients interact. SIENA has been designed to scale well in wide-area networks, such as the Internet, and supports an operational semantics based on the concepts of advertisements and subscriptions.

Events are defined as a set of typed attributes to which event filters and event patterns may be applied. Events are routed through a topology of interconnected servers, which may be organised according to either a client/server or a peer-to-peer architecture. The latter supports bi-directional information flow between servers based on standard network protocols, such as HTTP and SMTP. Advertisement and subscription information is used to optimise the network traffic between event servers according to the principles of downstream replication and upstream evaluation. As a result, event filters and patterns may be applied on any one of the event servers in a network. The location of the evaluation of a specific filter or pattern depends on the application.

SIENA has been enhanced to support nomadic event clients that use GPRS-based wireless networks to interact with event servers that are interconnected through a backbone of a fixed wide-area network. This mobility support is based on exploiting client proxies for maintaining event and subscription information on behalf of moving clients. Mobile clients use `moveIn` and `moveOut` operations, similar to the operations defined by JEDI, to disconnect and eventually re-connect to the proxies of an event service.

2.11 Elvin

The Elvin notification service has originally been developed to support a visualisation service for distributed systems [74]. This work, which has been undertaken at the University of Queensland in Australia, resulted in several versions of Elvin. The initial version, which was based on a simple, centralised architecture delivering all events to all consumers, has since evolved into Elvin3 [74] and Elvin4 [17, 35]. Elvin3, although still based on a single server architecture, provides a content-based subscription scheme and a mechanism for optimising event propagation, termed quenching. Elvin4 enhances its predecessor with a security framework, an architecture that scales beyond a single server, and support for mobile computing.

2.11.1 Elvin4 Architecture

Like its predecessors, Elvin4 supports a best-effort semantics and provides bindings for various programming languages. Currently available client libraries include support for Java, C/C++, Python, Smalltalk, Emacs LISP, and TCL. Producers and consumers connect to an Elvin4 server, which acts as router for disseminating events between its clients. Clients may either explicitly connect to a specific server or may exploit Elvin4’s automatic server discovery mechanism, which is based on multicast queries, to locate servers. Segall et al. [35] state that Elvin4 supports multiple servers in a local area environment and that these servers contain a mechanism for handing over client connections in order to facilitate load balancing. However, they do not explain how such servers cooperate to support interaction between clients connected to different servers.

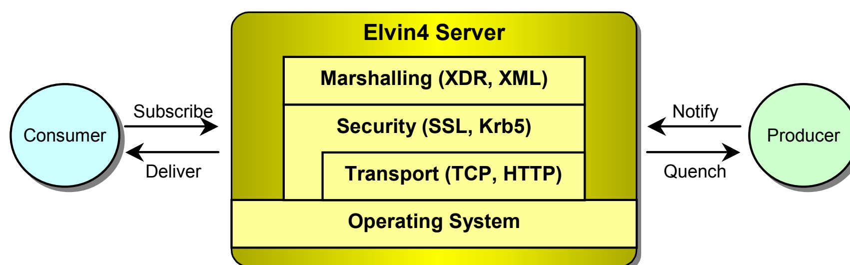


Figure 2.34. Overview of the Elvin4 architecture.

As illustrated in Figure 2.34, Elvin4 servers are based on a modular protocol stack comprising three layers, each supporting multiple protocols addressing different application requirements. Elvin4 events consist of sets of named values and a number of data types for

these values have been made available. Clients may use a set of simple arithmetic operators when defining subscriptions matching the values of the events in which they are interested.

Filtering

Elvin supports a filtering mechanism, known as quenching, that allows producers to determine whether any consumers have subscribed to their events. This may reduce the network traffic caused by events as producers can filter events for which they do not have a subscription. Essentially, quenching is a technique for discovering the recipients of specific events similar to approaches used in event services based on group communication as the underlying means for entities to interact. Such event services may exploit the membership management service typically provided by group communication to determine the set of subscribers.

Elvin servers maintain their subscriptions in a specific subscription database. Quenching requires servers to sent updates on this database to all quenching-enabled clients (Elvin4 allows lightweight clients to disable quenching) thereby causing additional overhead. However, it has not been outlined under with circumstances quenching is desirable, i.e., the reduction in event dissemination load exceeds the (potentially considerable) communication and computational overhead introduced by the quenching mechanism.

Security

Elvin4 supports a security framework for authorising event delivery. Producers supply a set of keys, which are distributed to consumers using mechanisms, such as shared file systems or directory services, provided by the application. Clients then use Elvin4’s security layer to encrypt keys when transmitting them to a server.

Figure 2.35 summarises this security framework and outlines how clients supply their keys to a server. Producers send their keys to a server before disseminating secure events. Subscribers apply a one-way hash function to their keys and send these transformed keys to the server. The sever authorises access to events by applying the same hashing algorithm to producer keys prior to matching them to the transformed subscriber keys. This approach allows keys being associated with either a connection to a specific client or an individual event. Moreover, a particular key may be used by a set of subscribers and producers ensuring authorised event dissemination within a group of clients.

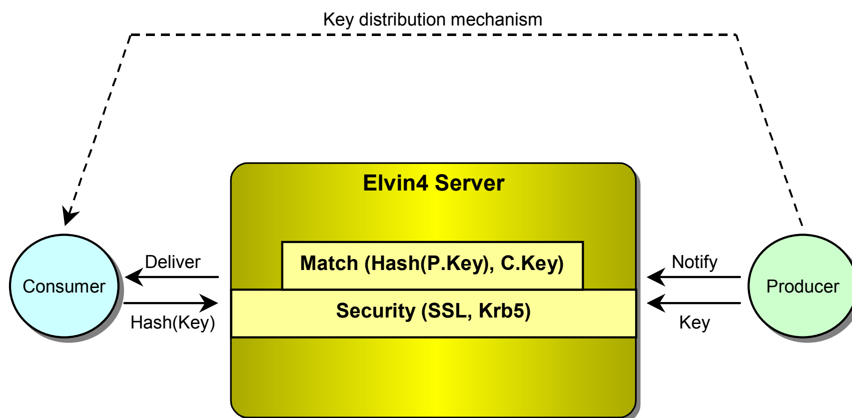


Figure 2.35. The Elvin4 security framework.

Mobility

Elvin4 supports mobility through the use of a proxy server that maintains a permanent connection to an event server on behalf of nomadic clients. The proxy server persistently stores events while a client is temporarily disconnected and forwards them when the client reconnects. Clients may specify a time to live for each subscription that can then be used by proxy servers to discard expired subscription information. This prevents proxy servers from buffering large numbers of events indefinitely.

One of the objectives of Elvin4 is to accommodate multiple, potentially mobile clients with identical sets of subscriptions on behalf of a particular user. For example, a user may use a desktop computer in conjunction with a notebook computer and a PDA. Each of these devices may act as an Elvin4 client wishing to access the user's subscriptions (or a subset of them). The concept of a session has been introduced to refer to such a group of subscriptions. Proxy servers maintain sessions storing relevant subscription information and events for their clients, which may access their session using a password protection scheme. This implies that clients must explicitly connect to a specific proxy server and nomadic clients are required to reconnect to the same proxy server each time they intend to access their events.

2.11.2 Summary

The Elvin notification service has evolved from version one to its current version, namely Elvin4. Elvin4 is a best-effort service that supports a content-based subscription scheme and provides client libraries for various programming languages. Event clients typically connect to

a specific Elvin4 server through which events consisting of set of named values are routed. Elvin4 servers contain a quenching mechanism that allows producers to determine whether any consumers have subscribed to their events. This mechanism enables producers to optimise event propagation by omitting unwanted events.

Elvin4 supports a security framework for authorising event delivery based on keys. Both consumers and producers use Elvin4's security layer to encrypt their keys prior to transmitting them to a server. A sever authorises access to its events by matching the keys supplied by a producer and its subscribers. This approach allows keys being associated with either a connection to a specific client or an individual event. Moreover, a particular key may be used by a set of subscribers and producers ensuring authorised event dissemination within a group of clients.

Elvin4 supports mobility through the use of a proxy server that maintains a permanent connection to an event server on behalf of nomadic clients and stores events while clients are temporarily disconnected. Proxy servers maintain sessions that store subscription information and events for a group of potentially mobile clients, which may access their session using a password protection scheme. This implies that clients must explicitly connect to a specific proxy server and are required to reconnect to the same proxy server each time they intend to access their events.

2.12 Perspectives

Existing research on event-based middleware for mobile computing has mainly focused on accommodating nomadic applications. Although there are some variations in the proposed approaches, the objective is essentially to support nomadic components that connect to an infrastructure network through dedicated access points and disconnect from the network while moving from one access point to another.

JEDI allows nomadic application components to connect to and disconnect from its event dispatcher using the `moveIn` and `moveOut` operations. The `moveOut` operation disconnects an entity from its current dispatching server, allowing it to move to another location and then to use the `moveIn` operation to reconnect to another dispatching server at a later time. The dispatching server buffers all relevant information while an entity is disconnected and forwards it upon reconnection.

SIENA's mobility support service provides comparable `moveIn` and `moveOut` operations allowing nomadic entities to connect to per-client proxy components that run on event servers

acting as access points. These proxies transparently manage (and synchronise) subscriptions and events on behalf of a moving entity. In contrast to JEDI, SIENA allows entities hosted by nomadic devices to interact with an event server through a wireless connection based on GPRS technology.

Mobile Push [16] proposes a similar approach to supporting nomadic application components in which entities disconnect from the event service infrastructure while moving. Nomadic entities may access the event service infrastructure either through fixed or wireless connections.

Bacon et al. [4] describe how mobility can be supported by the Cambridge Event Architecture exploiting a mediator to buffer and forward events for a temporarily disconnected mobile entity.

Huang et al. [6] discuss some of the issues arising when adapting a publish/subscribe system to a mobile environment and analyses the adaptation of a centralized event broker as well as a set of cooperating distributed event brokers. They suggest hosting event brokers on wireless access points that are interconnected through a fixed network.

Elvin4 [17] implements support for mobility through the use of a proxy server that maintains a permanent connection to the event server on behalf of nomadic entities. The proxy server stores events while an entity is temporarily disconnected until the client reconnects and allows entities to specify a time to live for each subscription to prevent large numbers of events being stored indefinitely. This approach is limited by the fact that entities must explicitly connect to a specific proxy server and nomadic entities are required to reconnect to the same proxy server each time they intend to access their events.

Although these middleware services support mobility, their main goal is to handle disconnection while an entity moves from one access point to another and consequently, they rely on the presence of a separate event service infrastructure. Significantly, none of these approaches provides for collaborative applications comprising entities that interact at a certain location without relying on a separate event service infrastructure.

The concept of using multiple, potentially overlapping zones to bound the scope of event dissemination, which has been proposed in ECO, has been adopted by Fiege et al. [75]. They introduce a component framework that allows individual entities to be bundled into units. The objective of this work is to provide an abstraction for encapsulating a collection of entities (and their events) into a higher-level component with well-defined interfaces that can be reused. Such components may propagate events reflecting a change to the sate of their collection of entities as a whole and may be composed to form other components. These

components may then be arranged in a hierarchical topology to form an event-based system. Furthermore, individual components may be part of a large-scale distributed system operating in a heterogeneous environment, therefore bridging the boundaries of different networks. In essence, this work uses scopes to provide an abstraction for encapsulation and reusability in a heterogeneous environment, and provides a means to map events across scope boundaries. In contrast, the work presented in this thesis exploits scopes as a natural way to identify events of interest for mobile entities. In particular, scopes may be used to define geographical areas within which certain events are valid; hence delivering them at the specific location where they are relevant.

CHAPTER 3: TAXONOMY OF DISTRIBUTED EVENT-BASED PROGRAMMING SYSTEMS

As event-based middleware is currently being applied for application component integration in a range of application domains, a variety of event services have been proposed to address different application requirements. This chapter applies the survey of existing event systems presented in chapter 2 to a taxonomy of distributed event-based programming systems [23]. A taxonomy is a classification that allows different examples of some generic type to be systematically arranged in groups or categorised according to established criteria [76]. The taxonomy presented here is structured as a hierarchy of the properties of a distributed event-based programming system and may be used as a framework to describe a distributed event-based programming system according to its properties.

The next section introduces our taxonomy of distributed event-based programming systems and related issues. Section 3.2 outlines the taxonomy in detail describing the identified event system properties and their relationships using figures and text as well as providing examples of existing event systems having these properties. Section 3.3 further illustrates specific event system properties by classifying a number of event systems selected from chapter 2 according to the taxonomy.

3.1 Introduction

The ultimate challenge of establishing a taxonomy is to identify the criteria according to which the area of interest is categorised and to arrange them systematically. Our taxonomy identifies a set of fundamental properties of event-based programming systems and categorises them according to the event model and event service criteria. The event service is further classified according to its organisation and interaction model, as well as other functional and non-functional features. These properties are then arranged in a hierarchical manner starting from the root of the taxonomy, which defines the relationship between event system, event service and event model. Each property is described providing corresponding terminology.

3.1.1 Exploiting the Taxonomy

In addition to providing a means of describing an event system, the taxonomy can be used to broadly summarise event systems and the taxonomy terminology can be used in the general discussion of event systems. Event systems can be classified according to the same taxonomy terminology and therefore, can easily be compared with each other or can be matched against system requirements. The taxonomy may serve as a basis for identifying the canonical combination of the properties of an event system required by a particular application domain, simply by applying the taxonomy to a number of existing event systems used in that particular application domain and by extracting the common combination of properties. This can be useful for the requirements and design engineering of a novel event system. Moreover, the taxonomy is expected to be utilised to identify novel combinations of the properties of event systems and consequently, may serve as a basis for discovering potential research issues to be addressed in future work.

3.1.2 Related Work

Our taxonomy presents a set of generic event system properties and hence can be used to classify virtually any distributed event-based programming system regardless of system scale or application domain. The taxonomy identifies a large variety of properties, including quality of service, mobility, and security, and describes these properties as well as their possible options in detail.

Existing work on describing event systems has focussed on providing a high-level reference model or on classifying event systems for a specific application area. Barrett et al. [77] present a framework for event-based software integration that provides a high-level model for identifying components commonly found at the heart of event-based software integration in large scale systems. This framework identifies the main components of an event system as informers, listeners, registrars, routers, message transformer functions, and delivery constraints. The framework describes the relationships among these components in detail using an object-oriented type model, but does not specify possible patterns of interaction between informers and listeners. Moreover, it does not explicitly identify functional event system features and omits non-functional features altogether.

The work of Rosenblum and Wolf [78] on a design framework for event observation and notification has focussed on supporting the construction of large-scale, event-based systems for the Internet. This framework comprises seven models, namely object model, event model, naming model, observation model, time model, notification model, and resource model, to

capture many of the design dimensions relevant to Internet-scale applications. Even though each of these models is discussed in detail, the overall number of properties according to which an event system may be classified is substantially smaller compared to the taxonomy presented in this thesis. This is due to the fact that this framework imposes certain constraints in order to specifically support Internet-scale event observation and notification and because certain issues, such as quality of service, mobility, and security, have not been considered.

3.1.3 Interpreting the Taxonomy

This taxonomy of distributed event-based programming models is presented using both figures and corresponding text. The figures outline the relationship among the fundamental properties of event systems and define the terminology to identify them. The text associated with each figure describes the corresponding properties in detail. The figures allow a taxonomy user to easily trace paths through the hierarchy to discover relevant properties. As summarised in Figure 3.1, the figures consist of nodes, one of which is the root node and some of which are leaves. Nodes are connected by directed paths. The directed paths are represented by a set of arrows describing the nature of the paths leaving a specific node. A set of dashed arrows leaving a specific node indicates that *exactly one* path has to be chosen when tracing through that node. Solid arrows indicate that *at least one* path has to be chosen, whereas double lined arrows indicate that *all possible paths* need to be selected. In order to apply the taxonomy to an event system, starting from the root node, a taxonomy user traces paths through the hierarchy selecting the connections that most accurately describe the event system until each selected path reaches a leaf. The terms associated with the nodes along a path describe a property of the event system.

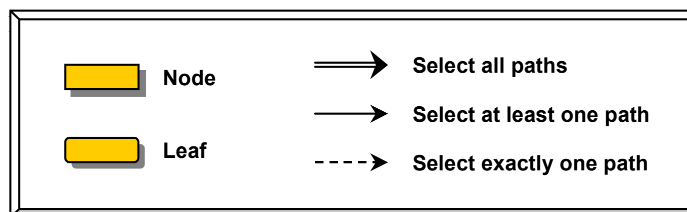


Figure 3.1. Taxonomy legend.

For example, Figure 3.21 shows that the features of an event service include both functional *and* non-functional features by using double lined arrows to describe the paths between the nodes. Hence, when tracing through the features node all paths, i.e., both of them, must be

selected to describe the corresponding properties of the event system. The solid arrows connecting the nodes in Figure 3.22 indicate that one kind of event propagation model can be associated with the functional features of an event service, although some event service may support both the sporadic and the periodic event propagation model. Therefore, either one or both paths must be traced. Figure 3.4 depicts that an event model can be characterised as *either* peer to peer, mediator or implicit. The dashed arrows connecting the nodes, which imply that exactly one path has to be chosen, illustrate this.

3.2 The Taxonomy

The root of the taxonomy, which is depicted in Figure 3.2, defines the relationship between event system, event service and event model. Figure 3.2 illustrates that every event system has both an event service and an event model. We define each of these terms as follows:

- An **event system** is an application that uses an event service to carry out event-based communication.
- An **event service** is middleware that implements an event model, hence providing event-based communication to an event system.
- An **event model** consists of a set of rules describing a communication model that is based on events.

We differentiate between event service and event model in order to capture the facts that an event model defines an application-level view of an event service and that a range of event services may implement a particular event model. Event models reflect the different usages for which they are intended. For example, the objectives of the Java AWT delegation event model differ substantially form those of the CORBA notification service, which leads to differences in the APIs that they provide. The goal of the event model of the CORBA notification service is to be extremely general-purpose and usable in virtually any domain. Consequently, it supports a wide range of features including typed and untyped event communication, as well as filtering and administrative capabilities. Moreover, a variety of quality of service properties, such as event reliability, connection reliability, event priority, and event delivery order, are supported to control the propagation characteristics of events. This is reflected in a fairly large and complex API. In contrast, the Java AWT delegation event model is intended for small-scale, centralised applications, such as graphical user interfaces, and therefore omits many of the features of the CORBA event model. This results in the API of the Java event model being much simpler than that of the CORBA event model.

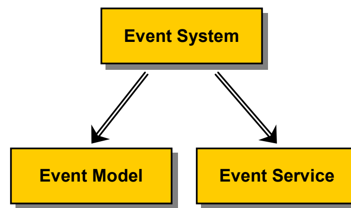


Figure 3.2. The root of the taxonomy.

The CORBA event model also serves as an example of an event model that was specified with the expectation of being implemented by a range of event services, and potentially being exploited in different application domains. The OMG leaves open the implementation of their model and therefore, leaves it to different vendors to provide implementations. Consequently, the CORBA event model has been implemented and extended by a number of commercial and academic organisations [79], [32], [31].

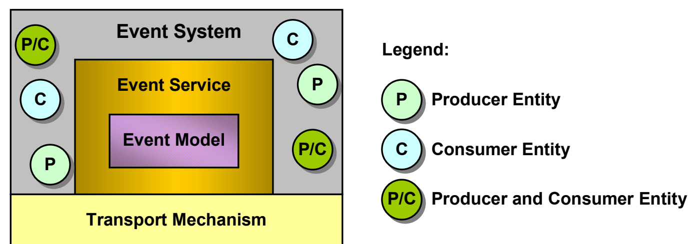


Figure 3.3. Event system overview.

The relationship between event system, event service and event model are summarised from the event system's perspective in Figure 3.3. Apart from depicting how an event system uses an event service that implements a particular event model, Figure 3.3 also outlines how event system and service map onto a transport mechanism and how applications use entities as hooks into the event service. Entities are the components of an application that produce and consume events, excluding components of the event service. An entity may play the role of either a producer or a consumer of events, or may act as both a producer and a consumer of events.

3.2.1 Event Model

The **event model** defines the manner in which an event service is made visible to the application programmer. It specifies the components of an event service to which the application programmer is explicitly exposed and which are used to subscribe to events and

to propagate them. In particular, the event model classifies the means by which the consuming entities of an application subscribe to the events in which they are interested and the means by which an application raises and delivers events, as well as the number and location of the components involved. As shown in Figure 3.4, we have identified three distinct categories of event model, which are peer to peer, mediator, and implicit.

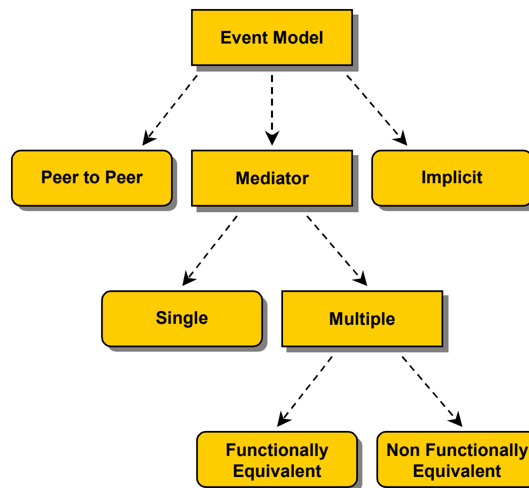


Figure 3.4. Event model categories.

Peer to Peer

A **peer to peer** event model allows consuming entities to subscribe at specific named producing entities *directly* and producing entities to deliver events to specific named subscribed entities *directly*. The Java distributed event model is based on a peer to peer event model and thus, allows a RemoteEventListener to subscribe to events by invoking a register method on an explicitly named EventGenerator. The simplified application shown in Figure 3.5 outlines a subscribing RemoteEventListener and an EventGenerator invoking the notify method on a subscribed RemoteEventListener using the RemoteEventListener's reference to deliver a specific event instance.

```
TheConsumerApplication {//the RemoteEventListener
  //subscribe to an explicit producer
  AnExplicitEventGeneratorRef = retrieveEventGeneratorRef();
  AnExplicitEventGeneratorRef.register(this);
  //delivery handler implementation
  notify(TheRemoteEventInstance) {
    processAnEvent(TheRemoteEventInstance);
  }
}

TheProducerApplication {//the EventGenerator
  //register method implementation
  register(RemoteEventListenerRef) {
    SubscribedRemoteEventListenerRef = RemoteEventListenerRef;
  }
  //raise an event
  AnEventInstance = new Event(someParameters);
  SubscribedRemoteEventListenerRef.notify(AnEventInstance);
  }
}
```

Figure 3.5. A producer and a consumer application using the peer to peer-based Java distributed event model.

Mediator

Event models utilising a **mediator** allow consuming entities to subscribe at a *designated* mediator and producing entities to deliver events to the mediator, which then forwards them to the subscribed entities.

The mediator sub-hierarchy explores the number and functionality of mediators in the event model. We differentiate between models utilising a **single** mediator and models exploiting **multiple** mediators. The CORBA event model[1] may use a single mediator (event channel) for propagating all events from suppliers to consumers. Multiple mediators are further divided into functionally equivalent and non-functionally equivalent mediators. In the former, all mediators are **functionally equivalent**. Thus, entities may subscribe or deliver events to any one of them. Such a mediator is called an event server in the SIENA model. SIENA may utilise a set of different event server topologies of which all but the centralised topology exploit multiple, functionally equivalent event servers. When mediators are **not functionally equivalent**, entities have to subscribe or deliver events to the correct mediator. For example,

[1] The CORBA specification allows its event model to utilise a single or multiple mediators. For the purpose of this example, we refer to a CORBA event model utilising a *single* mediator.

an application exploiting the CORBA event model[2] may use multiple event channels each propagating a different type of event.

The simplified application shown in Figure 3.6 outlines how both CORBA consumers and suppliers connect to the explicitly named event channel through which they intend to exchange events. Connected suppliers may raise events by pushing them to the event channel, which forwards them to all subscribed consumers by invoking their delivery handlers in turn.

```
TheConsumerApplication {
  //connect to an explicit event channel
  ConsumerAdmin = TheEventChannel.forConsumers();
  ProxyPushSupplier = ConsumerAdmin.obtainPushSupplier();
  ProxyPushSupplier.connectPushConsumer(TheConsumer);
}
TheConsumer {
  //delivery handler implementation
  push(TheRemoteEventInstance) {
    processAnEvent(TheRemoteEventInstance);
  }
}

TheProducerApplication {
  //connect to an explicit event channel
  SupplierAdmin = TheEventChannel.forSuppliers();
  ProxyPushConsumer = SupplierAdmin.obtainPushConsumer();
  ProxyPushConsumer.connectPushSupplier(TheSupplier);
}
TheSupplier {
  //raise an event
  AnEventInstance = new Event(someParameters);
  ProxyPushConsumer.push(AnEventInstance);
}
```

Figure 3.6. A producer and a consumer application using the mediator-based CORBA event model.

Implicit

An **implicit** event model lets consuming entities subscribe to particular event types rather than at another entity or a mediator. Producing entities generate events of some type, which are then delivered to the subscribed entities. The direct approach for CEA source objects to disseminate events to client objects, described by Bacon et al. [33], is based on an implicit

[2] The CORBA specification allows its event model to utilise a single or multiple mediators. For the purpose of this example, we refer to a CORBA event model utilising *multiple* mediators.

event model. Figure 3.7 shows a simplified version of an active badge application using direct CEA. The consumer subscribes by invoking a register method provided by a *local* library passing the event type of interest as well as a reference to its delivery handler. The producer declares its event type and subsequently raises events of this type by invoking a signal method provided by a *local* library. The event service delivers events to all registered consumers by calling their delivery handlers.

```
TheConsumerApplication {
  //subscribe to an event type
  template = Badge_Seen(17, 29);
  EventClient.Register(EventHandler, template);
  // deliver handler implementation
  EventHandler(TheRemoteEventInstance) {
    processAnEvent(TheRemoteEventInstance);
  }
}

TheProducerApplication {
  //specify the event type
  Badge : INTERFACE = Seen : EVENTCLASS [badge  : BadgeId;
                                         sensor : SensorId];
  END.
  //raise an event
  e = Badge_Seen(17, 29);
  EventSource.Signal(e);
}
```

Figure 3.7. A producer and a consumer application using the implicit Direct CEA.

Discussion

An event system exploiting either a peer to peer or a mediator-based event model allows its entities to interact by invoking remote methods *directly* on each other or on one or more mediators respectively whereas entities of an event system with an implicit event model interact by subscribing and delivering events *locally* using event types.

Significantly, these approaches differ in the way identifiers to the components exposed to the application programmer are obtained and maintained. Peer to peer and mediator-based event models require the application programmer to obtain identifiers to explicitly named producers and mediators respectively, usually by means of exploiting a lookup table or a naming service, and to maintain them. Every consumer of an event system utilising a peer to peer-based event model is required to obtain the identifier of each producer in which it is interested, i.e., the application programmer must ensure a consumer subscribes to the correct set of producers, and to maintain these identifiers during their lifetime. Similarly, entities of an event system utilising a mediator-based event model need to acquire the

identifiers of the mediators involved, i.e., the application programmer must track the identifiers to the mediators to which a specific entity needs to connect. However, mediator-based event models are likely to obtain and maintain a smaller number of different identifiers compared to peer to peer-based event models. There are likely to be significantly fewer mediators in an event system than producers and their quantity is unlikely to change over time[3], certainly compared to the number of producers as they may be created frequently providing services for a limited period of time. Therefore, the number of the components explicitly exposed to the application programmer is expected to be significantly smaller in a mediator-based event model compared to a peer to peer-based event model. In contrast, the application programmer in an event system with an implicit event model is not required to acquire any identifiers to entities or mediators at all. The application programmer does not need to explicitly identify the producers with which a consumer needs to communicate as consumers subscribe to producers transparently using event types. This requires a more sophisticated event service as it is responsible for locating peers, maintaining the corresponding identifiers, mapping event types to identifiers, and for providing a means to define and check the type of events.

Most significantly, the event model exploited by an event system affects one of the main concepts of event-based communications, namely the degree of anonymity among the entities in the system. The means by which consuming entities subscribe to the events in which they are interested and by which events are propagated and delivered influences the degree of anonymity among them. The peer to peer approach permits specific named entities to interact directly with each other. Consequently, entities are not anonymous to each other. Mediator-based event models, where entities register with one or more mediators, provide a degree of anonymity where entities are anonymous to each other but known to the mediator(s). The implicit approach allows entities to interact anonymously. Such entities are anonymous to each other and are only known by the event service that implements the mapping of event types to entities.

3.2.2 Event Service

This section deals with the classification of the properties of an event service. As Figure 3.8 shows, we divide the properties of an event service into three distinct categories. The

[3] An event system may exploit a single mediator whose reference characteristically remains unchanged, assuming the absence of failure, during the lifetime of the system.

organisation sub tree focuses on the distribution of the entities and the components of the middleware and on the fashion in which the components that comprise an event service cooperate. The interaction model defines the communication path over which producing and consuming entities communicate with each other. The feature sub hierarchy addresses the other functional and non-functional features proposed by an event service.

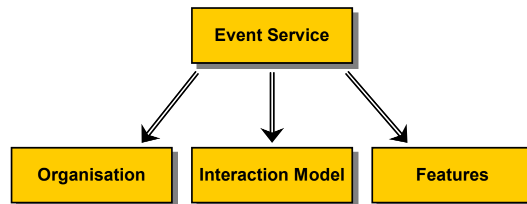


Figure 3.8. The event service.

Organisation

As summarised in Figure 3.9, the **organisation** sub tree classifies an event service as either centralised or distributed according to the location of the event system's entities. These two sub categories are further divided exploring the location of the event service's components.

The entities of an event system can be either centralised or distributed according to their location. The entities of an event system are **centralised** if they only reside in the same address space on the same physical machine. In contrast, if the entities of an event system are **distributed** they may be located in different address spaces possibly on different physical machines.

Whether the entities of an event system are centralised or distributed, the middleware can be either collocated or separated.

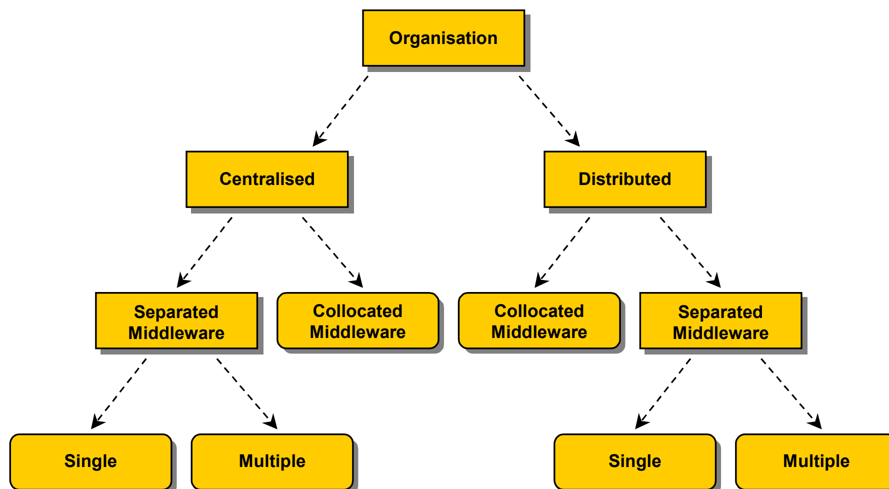


Figure 3.9. Event service organisation.

Collocated Middleware. The event service is collocated with the entities, if it resides only in the same address space(s) on the same physical machine(s). As illustrated in Figure 3.10, the organisation of a centralised event service with collocated middleware results in both the entities and the middleware being located exclusively in the same address space. No part of the event system resides outside the implicit single address space. This organisation may be used for small-scale applications consisting of a relatively small number of entities, such as graphical user interfaces. For example, the Java AWT delegation event model is implemented by the Java Virtual Machine (JVM) to connect the graphical components of an application sharing their address space with the middleware. Another event service that may be used in a similar fashion is provided by the C# programming language [25]. In contrast, the organisation of a distributed event service with collocated middleware results in the middleware being distributed with the entities, each entity using the part of the middleware that is local to it. Figure 3.11 shows the organisation of a distributed event service with collocated middleware, which may include an arbitrary number of address spaces. This organisation has been adopted by mSECO, an event service implementing the ECO event model. mSECO is implemented as a library that is collocated with each entity. Notably, mSECO is exclusively located in the same address spaces as the entities. However, the address spaces in which the entities reside may or may not be located on different physical machines.

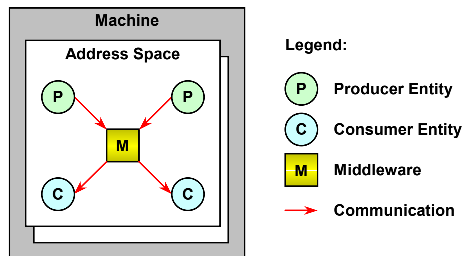


Figure 3.10. Centralised event service with collocated middleware.

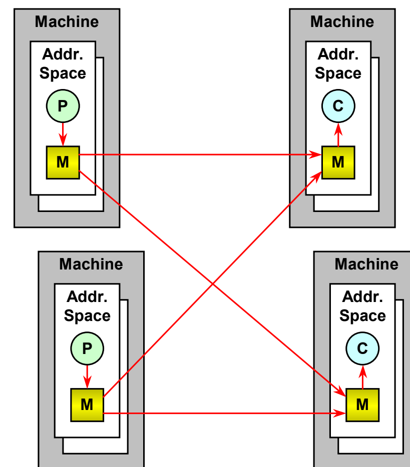


Figure 3.11. Distributed event service with collocated middleware.

Separated Middleware. In this case, the event service is at least partially located in one or more separate address spaces possibly on different physical machines. We divide separated middleware into two categories depending on the partitioning of the middleware. Figure 3.12 depicts an event service with separated **single** middleware, whose entities are centralised and whose middleware is located in a separate address space. This organisation uses exactly two separate address spaces, one including the entities and the other containing the middleware. The two address spaces may reside on the same or on two different physical machines.

Figure 3.13 illustrates a distributed event service with separated single middleware, whose entities are distributed and whose middleware is located on a single machine. This organisation may involve a large number of address spaces and possibly physical machines, depending on the location of the entities and the middleware. However, all the address spaces may reside on a single physical machine. A CORBA event service providing a single event channel[4] serves as an example of such an organisation. Its entities typically reside in different address spaces distributed over multiple physical machines using an event channel located on another machine. However, the address space in which the event channel resides may be located on the same physical machine as some of the entities' address spaces.

[4] The CORBA event service may utilise one or more event channels. For the purpose of this example, we refer to a CORBA event service utilising a *single* event channel.

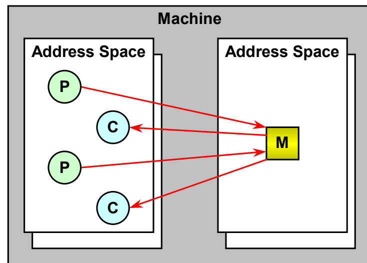


Figure 3.12. Centralised event service with separated single middleware.

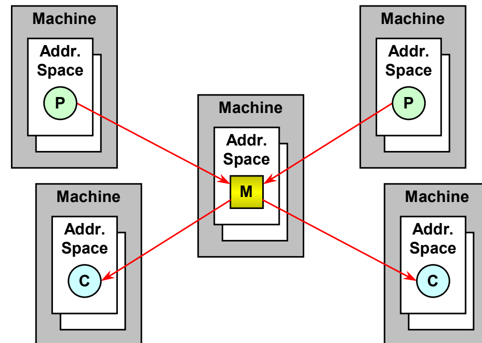


Figure 3.13. Distributed event service with separated single middleware.

Figure 3.14 and Figure 3.15 show event services with separated **multiple** middleware, whose middleware is distributed over a set of cooperating address spaces possibly on different physical machines, for a centralised and a distributed organisation respectively. Figure 3.15 also illustrates that some of the middleware's address spaces may be located on the same machine as some of the entities. This also applies for centralised entities with separated multiple middleware. We admit the possibility of an organisation of centralised entities with separated multiple middleware although we cannot provide an example for such an organisation. SIENA, which uses an organisation as shown in Figure 3.15, proposes a set of middleware topologies, called server topologies, of which all but the centralised topology use middleware that is distributed over a set of cooperating machines.

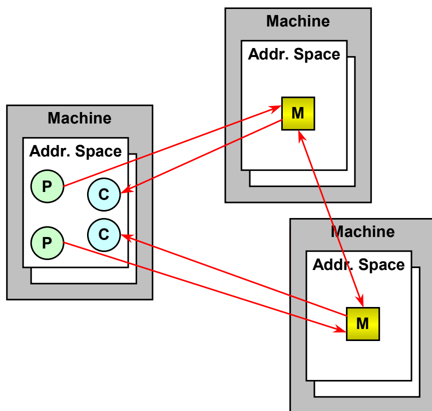


Figure 3.14. Centralised event service with separated multiple middleware.

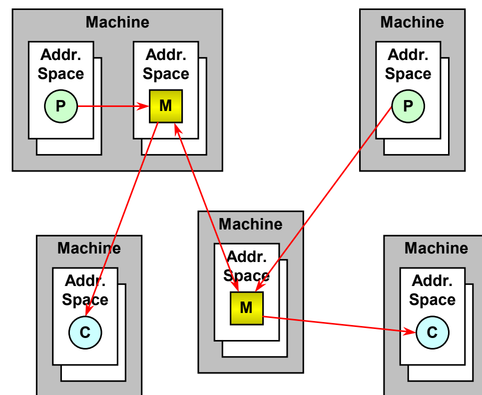


Figure 3.15. Distributed event service with separated multiple middleware.

Discussion. The organisation adopted by an event service has a major impact on issues related to the scalability of a system, its behaviour in the presence of failed components, and on the mechanism for communication between entities and the middleware. Traditionally, approaches containing centralised middleware components are more likely to experience performance bottlenecks with increasing scale and tend to suffer more in the presence of failures than distributed approaches. The use of middleware located in multiple address spaces allows the distribution of the communication load reducing the risk of performance bottlenecks. Instead of having middleware located in a single address space handling all the communication between the entities in an event system, middleware distributed over multiple address spaces may divide the load. Exploiting middleware distributed over multiple address spaces also avoids potential single points of failure in the system. For example, if the middleware in the organisations illustrated in Figure 3.10, Figure 3.12, and Figure 3.13 fails none of the entities in the corresponding systems will be able to communicate. In contrast, a middleware component failing in one of the organisations depicted in Figure 3.11, Figure 3.14, and Figure 3.15 has a less devastating effect on an event system allowing the entities to communicate even in the presence of failure. Significantly, this depends on the middleware being located in multiple address spaces and not on the distribution of the entities in a system.

The organisation of an event service also affects the mechanism through which entities communicate with the middleware. Approaches where entities and middleware reside in different address spaces distributed over different physical machines require a mechanism that supports communication across the boundaries of address spaces and network connections. A much simpler inter-process communication mechanism may be sufficient for organisation where entities and middleware reside in different address spaces on the same physical machine. Entities and middleware sharing an address space may communicate using a programming language specific mechanism, such as procedure call or method invocation.

This taxonomy may serve as a basis for identifying the combinations of event system properties that are well suited as well as the combinations that are less suited or even incompatible. For instance, mediator-based event models map well onto event service organisations with separated middleware. Separated middleware residing in an independent address space may naturally implement a mediator to which producing and consuming entities may subscribe. Peer to peer and implicit event models are well suited for organisations with collocated middleware. These organisations allows entities to directly connect to each other using interfaces specified by the collocated middleware, which provides a means for mapping events and their types to entities. In addition, the centralised

organisation with collocated middleware may map onto mediator-based event models as the collocated middleware may implement a mediator for entities to interact. In contrast, combinations, for example based on separated middleware and peer to peer event models, are suited to a lesser extend as peer to peer models imply that entities interact directly.

Interaction Model

The interaction sub tree classifies an event service according to the **interaction model** used by the event system. Generally, the interaction model defines the communication path over which event communication between producing and consuming entities takes place. It defines the number of intermediate middleware components involved and the manner in which intermediates cooperate to route events from producers to consumers. Compared to the organisation model, which focuses on the distribution of the entities and the middleware of an event system describing the static view of an event service, the interaction model describes the information flow in a event system. Hence, it describes the dynamic aspect of an event service.

As Figure 3.16 depicts, we divide the interaction model into two main categories, namely intermediate and no intermediate, exploring whether and how many intermediate middleware components an event passes through.

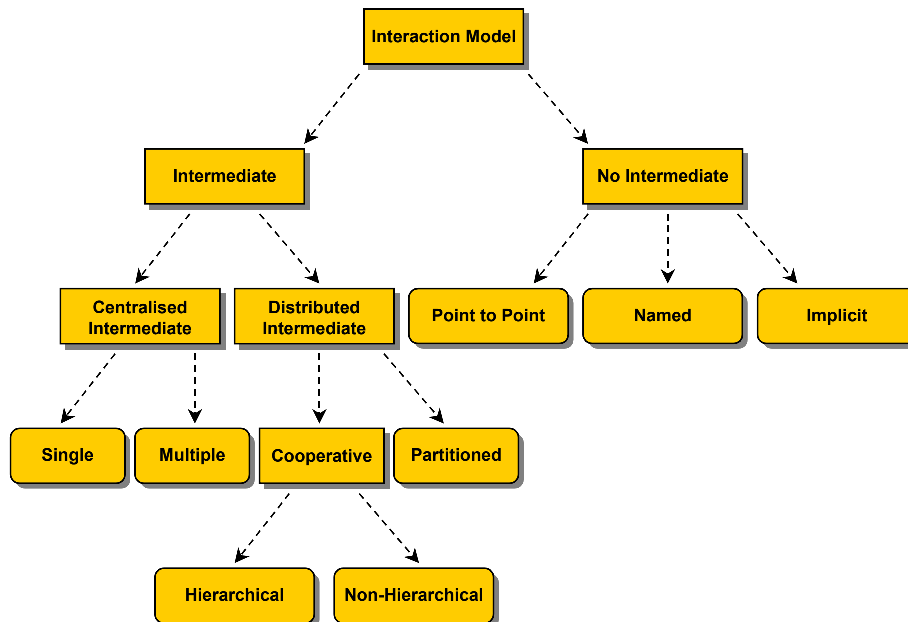


Figure 3.16. Event service interaction model.

No Intermediate. The communication path over which event communication between producing and consuming entities takes place does not include separated intermediate middleware components. Producer and consumer entities communicate with each other through the middleware collocated with each entity. As Figure 3.17 illustrates, events that are routed from producers to consumers pass through the respective collocated middleware, but not through any intermediate middleware component.

Intermediate. The communication path over which event communication between producing and consuming entities takes place includes at least one separated intermediate middleware component. Events that are routed from producers to consumers pass through one or more intermediate middleware components.

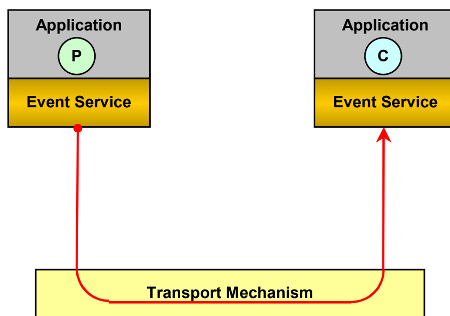


Figure 3.17. No intermediate.

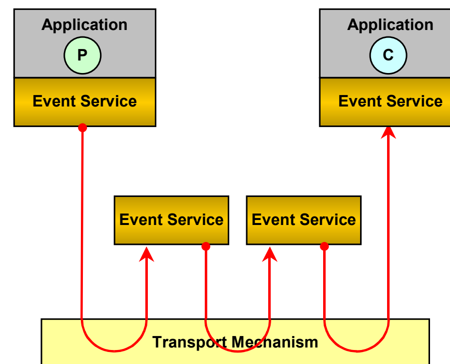


Figure 3.18. Distributed intermediate.

The intermediate interaction model is divided into two sub categories according to the number of intermediate middleware components in the communication path between producing and consuming entities. In the **centralised intermediate** model, the communication path includes a single intermediate middleware component. In contrast, the **distributed intermediate** involves two or more intermediates through which events are routed from producers to consumers. Figure 3.18 depicts the distributed intermediate interaction model with a communication path that includes two distributed intermediates.

Both centralised and distributed intermediates can be divided further. We classify centralised intermediates according to their number as an event service may exploit single or multiple centralised intermediates.

All communication paths between producing and consuming entities may include the same **single** centralised intermediate. An event system using this interaction model includes exactly one centralised intermediate. In contrast, an event system may exploit **multiple** centralised intermediates. Producers and consumers may be divided into groups and all

communication paths between the producing and the consuming entities of each group may include a centralised intermediate that is exclusive to the group. This results in an event system using several centralised intermediates, the number of which corresponds to the number of entity groups. Multiple centralised intermediates may be used to support groups of entities that share a common interest. The common interest of an individual group may be expressed by a specific type of event that is exclusively handled by a particular centralised intermediate. For example, the CORBA event service may utilise multiple centralised intermediates implemented as event channels. Each channel may handle a specific type of event exclusively. Producing and consuming entities intending to communicate using a specific event type connect to the corresponding event channel, therefore defining the communication path over which event communication takes place. Alternatively, the CORBA event service may utilise a single centralised intermediate implemented as a single event channel through which all events are routed. Figure 3.19 and Figure 3.20 illustrate the single centralised intermediate and the multiple centralised intermediate interaction model respectively. Figure 3.20 shows two groups of entities, each comprising of a producer and a consumer exclusively using a single centralised intermediate through which events are routed. The communication path associated with one group is outlined as solid arrows and the communication path associated with the other is depicted as dashed arrows.

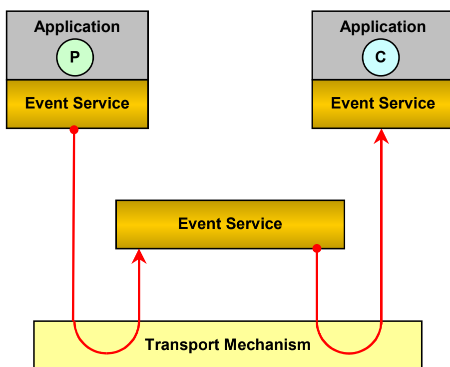


Figure 3.19. Single centralised intermediate.

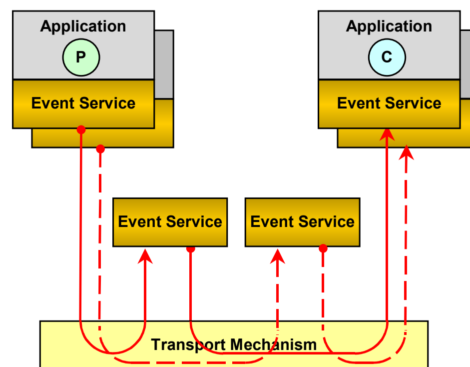


Figure 3.20. Multiple centralised intermediate.

We classify distributed intermediates as partitioned or cooperative according to the fashion in which intermediates cooperate to route events from producing to consuming entities.

Generally, the distributed intermediate interaction model includes two or more intermediate middleware components in the communication path between consumers and producers. An event service implementing the **partitioned distributed intermediate** interaction model

consists of one or more independent groups of intermediates, each group handling only a specific type of event. Entities sharing a common interest need to connect to the group that handles the type of event that corresponds to their common interest. The CORBA event model specification proposes to chain different implementations of event channels, acting as a group of partitioned distributed intermediates, in order to combine non-functional features supported by individual event channels.

In contrast, **cooperative distributed intermediates** do not form independent groups, all intermediates cooperate to route events from consumers to producers. Entities connect to the most convenient, e.g., physically closest, intermediate. Each intermediate manages the events of the entities physically connected to it and cooperates with other intermediates to route them to remote entities. Cooperative distributed intermediates cooperate with each other either in a **hierarchical** or in a **non-hierarchical** manner. JEDI proposes a hierarchical structure of cooperative distributed intermediates, called dispatching servers. Dispatching servers are interconnected in a tree topology through which events are routed. Entities may connect to any dispatching server, each of which forwards the events it receives from the producing entities connected to it to its parent and to its descendants to route them to all interested consumers. SIENA describes four different topologies of cooperative distributed intermediates. One of them serves as an additional example of hierarchical cooperative distributed intermediates, another two, namely the acyclic and the general peer to peer topologies, illustrate non-hierarchical cooperative distributed intermediates.

We sub divide the interaction model that does not include intermediate middleware components into three categories according to the means by which entities address each other. These interaction models are called point to point, named, and implicit.

Producer and consumer entities may communicate directly with each other in a **point to point** fashion, using explicit entity addresses, which are provided by the application. The middleware uses explicit entity addresses and a unicast communication pattern when routing events from producing to consuming entities. The Java distributed event model allows producers to route events to the subscribed consumers using the explicit consumer addresses provided by the application.

Producer and consumer entities may communicate directly with each other using a **name** service to map event descriptions, such as event types, to entity addresses provided by the application. The middleware uses either a unicast or a multicast communication pattern to route events from a producer to the interested consumers. uSECO uses an name service, called the Application Instance Repository (AIR), to resolve the addresses of the entities that are interested in a specific event type and a unicast communication pattern to route events.

Producer and consumer entities may communicate directly with each other using an **implicit** means to map event descriptions to entity addresses provided by the application. The middleware uses a multicast communication pattern when routing events from producers to consumers. mSECO, a multicast version of the uSECO event service, does not rely on an AIR since it uses an implicit means, based on generating addresses from event descriptions, to map events to the multicast addresses representing the interested consumers.

Discussion. Mediator-based event models map naturally onto interaction models that include intermediate middleware components. For example, interaction models using either multiple centralised or partitioned distributed intermediates may implement event models that include multiple non-functionally equivalent mediators. These event models expose mediating application components to the application, which must ensure entities subscribe to the correct intermediate middleware component. Both hierarchical and non-hierarchical cooperative distributed intermediates may implement multiple functionally equivalent mediators whereas a single centralised intermediate may implement an event model based on a single mediator. Both the named and the implicit interaction model are appropriate for implicit event models, since neither of them relies on intermediates and because implicit event models do not prohibit the use of middleware components providing naming services. The peer to peer event model exposes entities explicitly to the application. It is therefore best implemented by a point to point based interaction model using these entity addresses to route events from producers to consumers.

There are numerous possible combinations of interaction and organisation models as many organisations are appropriate for different interaction models. For example, both centralised and distributed organisations with separated middleware are suitable for interaction models whose communication paths between producers and consumers involve intermediate middleware components. Distributed organisations with collocated middleware may be combined with interaction models that do not rely on intermediates. Centralised organisations with collocated middleware may possibly be combined with every identified interaction model. Although centralised collocated organisations may be best suited for the single intermediate interaction model as its middleware component maps naturally onto a single intermediate, it is also appropriate for the implicit interaction model with its middleware component implementing a means to map event types to entity addresses.

Features

The **features** supported by an event service can be classified as either **functional** or **non-functional**. Both the functional and the non-functional features are sub divided into a set of

relevant issues, each of which will be discussed in turn in the remainder of this section. The sets of functional and non-functional features that may be supported by an event service are summarised in Figure 3.21.

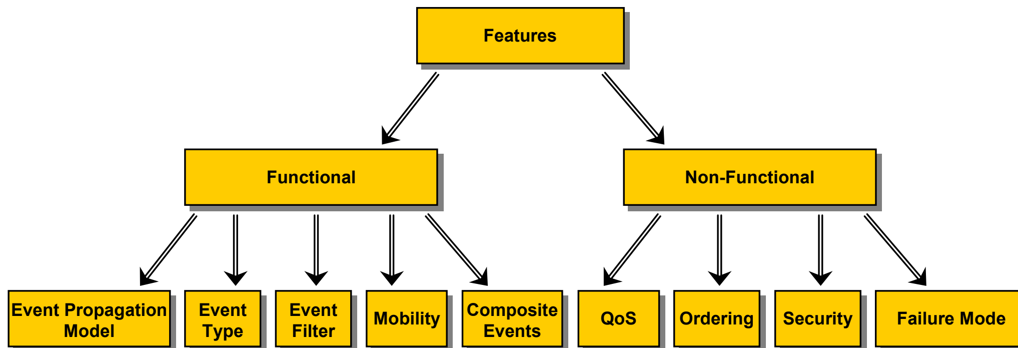


Figure 3.21. Event service features.

Functional Features

Event Propagation Model. Events are propagated and delivered by an event service according to an event propagation model. Figure 3.22 depicts the event propagation model sub hierarchy and shows how the event propagation model is divided into two categories describing sporadic and periodic event propagation. Both sporadic and periodic event propagation can be based either on the push or the pull model. The sporadic push model is considered the traditional event propagation model and is therefore most likely to be supported by an event service. However, an event service may support several of the propagation models summarised in Figure 3.22.

Sporadic event propagation models propagate events only if the relevant state of the producer has changed. **Periodic** event propagation models propagate events periodically, even if no state change has occurred since the last event.

Event propagation based on the **sporadic push** model is considered the typical event propagation model. Event propagation is producer driven and producers propagate events as they are generated. The sporadic push model is supported by many event models including the Java AWT delegation event model and CORBA-based event models, such as CONCHA.

Event propagation based on the **sporadic pull** model is also known as event polling. Event propagation is consumer driven as consumers poll producers for available events. Event producers propagate events in response to requests from consumers. Among others, this propagation model is supported by the CORBA notification service.

Event propagation based on the **periodic push** model is well suited for “heartbeat” or “watchdog” mechanisms as well as for disseminating events according to a predefined schedule. Event propagation is producer driven and producers propagate events periodically. The TAO real-time CORBA event service uses the periodic push propagation model as a means to statically schedule event propagation and subsequently to reserve the required resources for events that have hard real-time delivery deadlines.

Event propagation based on the **periodic pull** model represents traditional polling. Event propagation is consumer driven, event consumers poll event producers periodically. Event producers propagate events in response to requests from consumers.

Periodic event propagation models imply that events with identical content may be propagated as the state of a producer that describes such content may have remained unchanged since the previous event was propagated. We argue that periodic events conform to the previously given definition of events when considering the passing of time as a change to a producer’s state even though periodic events may not contain an explicit description of time.

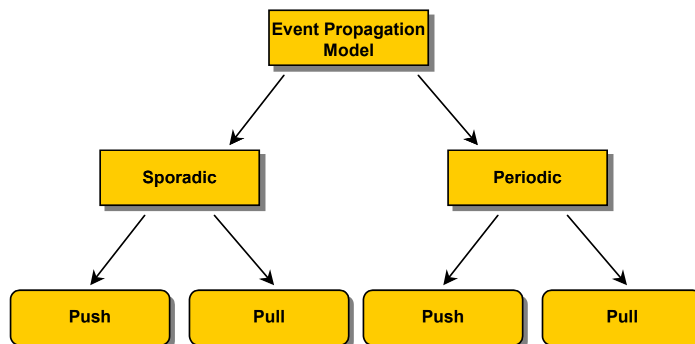


Figure 3.22. Event propagation model.

Event Type. Events propagated by an event service can be classified according to their structure and hence are said to be of a specific event type. As outlined in Figure 3.23, we differentiate between generic and typed events.

The information that constitutes a **generic** event, which is also known as an un-typed event, is a data blob with an *implicit structure*. The structure is neither recognised nor interpreted by the event service. The CORBA event service is one of the few event services that supports propagation of generic events.

In contrast, the information that describes **typed** events includes an *explicit and expressive structure* that may be recognised and interpreted by the event service. Typed events enable the use of event filters.

Event types are represented by a structure with varying **expressive power**. Expressive power of event types describes the variety of information that they can comprise. The expressive power of the structures outlined in Figure 3.23 increases from left to right.

The structure that represents and event type is either **fixed** or **application specific**. The former is predefined by the event service whereas the latter may be defined by the application.

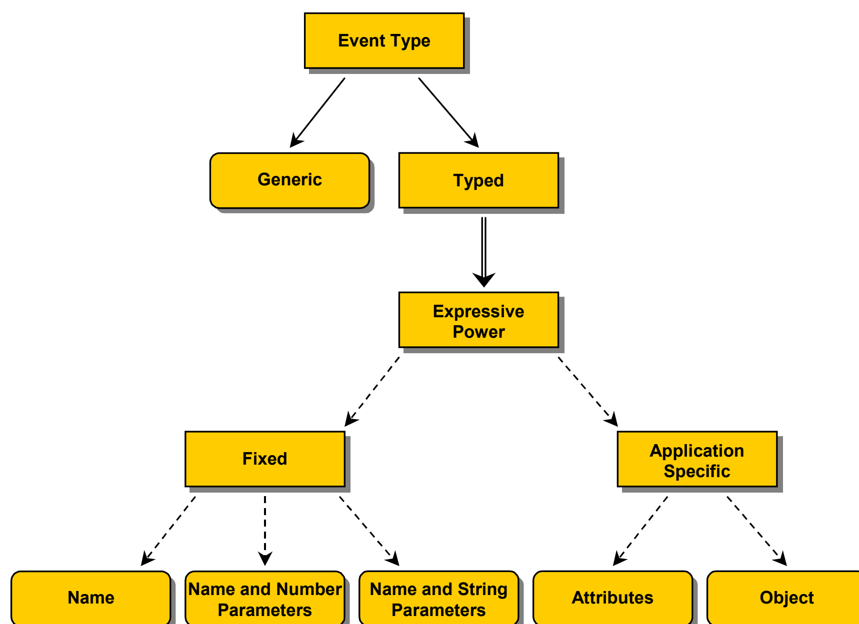


Figure 3.23. Event type.

Both fixed and application specific structures can be sub divided. Fixed structures consist either of a name, a name and number parameters, or a name and string parameters. A **name** usually consists of a single string. The **name and string parameters** structure therefore consists of a set of strings. The first string representing the event name and the remaining strings representing the event parameters. JEDI uses an event structure that is similar to function calls consisting of a name and a set of string parameters. The **name and number parameters** structure consists of a single string and a set of numbers. The string representing the event name and the numbers representing the event parameters. The version of CEA described by Bacon et al. [33] supports typed events that consist of a

structure consisting of a name and a set of number parameters. Application specific structures consist of either attributes or an object. The **attributes** structure consists of a set of attributes in which each attribute is a triple of name, type and value. The CORBA notification event service supports a general event structure consisting of attributes. The **object** structure consists of a programming language specific object including a set of attributes. One of the key properties of ECO is its support of events in the form of specific application defined objects.

Event Filter. Event filters control the propagation of events by allowing consumers to subscribe to the exact subset of the events in which they are interested. Events are matched against filters and are only propagated if the match produced a positive result. Figure 3.24 shows the properties according to which we classify event filters.

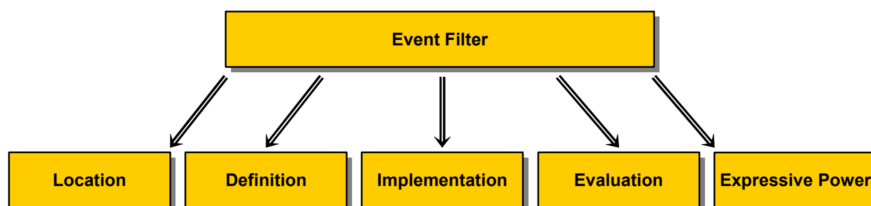


Figure 3.24. Event filter.

Event filters must be evaluated at a particular **location**. If supported, event filters may be evaluated at the consumer side, the producer side or at the intermediate. Furthermore, a set of event filters may be evaluated sequentially at more than one location, thus they may be applied at any combination of consumer, producer, and intermediate. Figure 3.25 summarises all possible combinations of event filter locations.

Nowhere. Filters are not supported. The CORBA event service is an example of an event service that does not support event filters.

Producer. Filters are evaluated at the producer side. This minimises the use of network bandwidth and consumer processing overhead as events are filtered as close to the producer as possible. SECO serves as an example of an event service that supports producer side filtering.

Consumer. Filters are evaluated at the consumer side. This allows an implementation of a precise matching algorithm as the required set of events is typically well known at the consumer side. The Java distributed event model allows filters to be applied at the remote event listener.

Intermediate. Filters are evaluated at the intermediate. This is a natural location for service wide filters (as well as quality of service properties) since all events are propagated through the intermediate.

Producer and Consumer. Filters are evaluated at the producer and the consumer side. ECO supports filters in the form of pre- and post constraints, which may be applied at the producer and the consumer side respectively.

Producer and Intermediate. Filters are evaluated at the producer side and at the intermediate.

Consumer and Intermediate. Filters are evaluated at the consumer side and at the intermediate. In addition to allowing filtering at the remote event listener, the Java distributed event model supports optional event adapters at which filters may be applied as well.

Producer, Consumer and Intermediate. Filters are evaluated at the producer and the consumer side, as well as at the intermediate. The CORBA notification service supports filtering in a hierarchical manner that allows filters to be evaluated at the producer and the consumer side, as well as at intermediates.

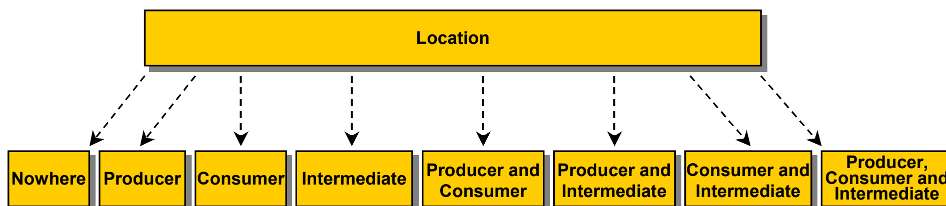


Figure 3.25. Event filter location.

As shown in Figure 3.26, event filters can be **defined** by the application by using a **constraint language** that is specified as part of the event service or by using the features of an application **programming language**. The CORBA notification service specifies a constraint language that allows applications to use constraint expressions to define event filters.

When using a programming language to define event filters, applications may use a **subset** of the types, operators, and combinators supported by the programming language or may be permitted to use all types, operators, and combinators supported by the **language**. SIENA limits applications to using a specific subset of the types, operators, and combinators available whereas SECO allows them to use all available types, operators, and combinators.

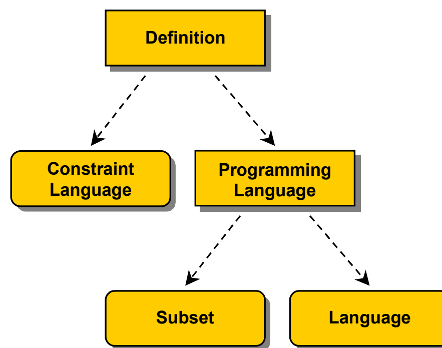


Figure 3.26. Event filter definition.

Figure 3.27 summarises possible **implementations** of event filters within the event service. An event filter can be implemented as either a character **string**, a **function**, or an **object**. Character strings can provide a textual representation of filter expressions that are typically parsed by an event service applying them. Filters that are implemented as functions are applied by executing these functions. Object class filters must be instantiated before they can be applied by invoking a method of the object. Both the CORBA notification service and SIENA implement event filters as strings that are interpreted at run time whereas SECO filters are implemented as objects providing an `evaluate()` operation.

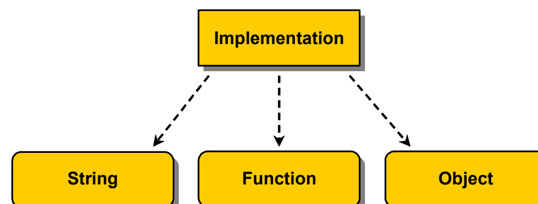


Figure 3.27. Event filter implementation.

Event filters are **evaluated** by the event service to determine the list of interested subscribers. As shown in Figure 3.28, event filters are evaluated at a particular **time** using a specific **mechanism** to match events against filters.

The evaluation mechanism is divided into two sub categories depending on whether filter specifications are **interpreted** or **compiled**. The former are characteristically evaluated using an event model specific interpretation mechanism while the latter can be evaluated using operations provided by the programming language. Both interpretable and executable filters are either generated by a **pre-processor** or are **implicitly** provided by the application. The

CORBA notification service specifies a constraint language that allows applications to implicitly provide filter expressions that are interpreted by the evaluation mechanism.

Event filters are evaluated either at **subscription** time or at event **propagation** time. Evaluating filters at subscription time may be useful when matching parameters describing the current context of the subscriber that are only relevant at that point in time or when matching pre-constraint filters. Such pre-constraint filters may assess the availability of resources, authenticate a connection, or process admission control. However, event services, including the CORBA notification service, SIENA, and Elvin, traditionally evaluate event filters at event propagation time when the actual list of interested subscribers can be determined.

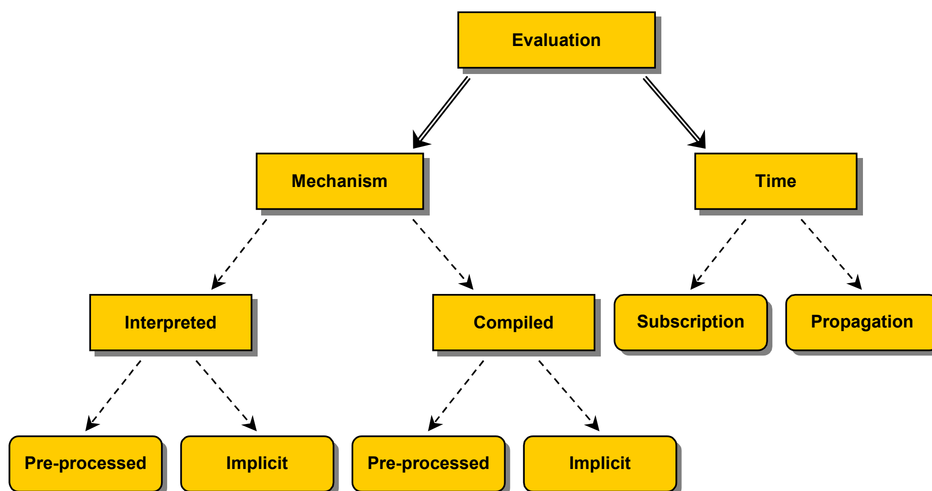


Figure 3.28. Event filter evaluation.

Figure 3.29 summaries issues related to the **expressive power** of event filters. Event filters may be defined using an expressive structure that is described using a set of types, operators, and combinators.

The structure enclosed in an event filter may contain a set of types with varying expressive power. These sets are either **implicit** or **predefined** by the event service and their expressive power generally increases with the number of the types they comprise. While both implicit and predefined sets can contain one or more types, predefined sets are typically larger and hence more expressive than implicit sets. JEDI and CEA are examples of event models supporting implicit types. Both of them support string types while CEA provides a second implicit type, namely number. In contrast, SIENA provides predefined sets comprising a larger number of types.

An event filter may contain a set of **operators** with varying expressive power. From left to right, the sets outlined in Figure 3.29 are supersets of each other and hence increase in their expressive power. The representations may support **equality** and inequality operators, less than and greater than **magnitude** operators that may be combined with equality operators, or magnitude operators that can be combined to form **range** operators. JEDI and CEA only support equality operators whereas SIENA supports equality, magnitude, and range operators.

An event filter may employ a set of **combinators** with varying expressive power that may be used to combine terms including types and operators. The expressive power of the set of combinators outlined in Figure 3.29 increases from left to right. The structure may **not** contain any combinator or may contain either a single **implicit** combinator or an **arbitrary** number of combinators. CEA supports an implicit conjunctive combinator while SIENA provides a range of arbitrarily applicable combinators.

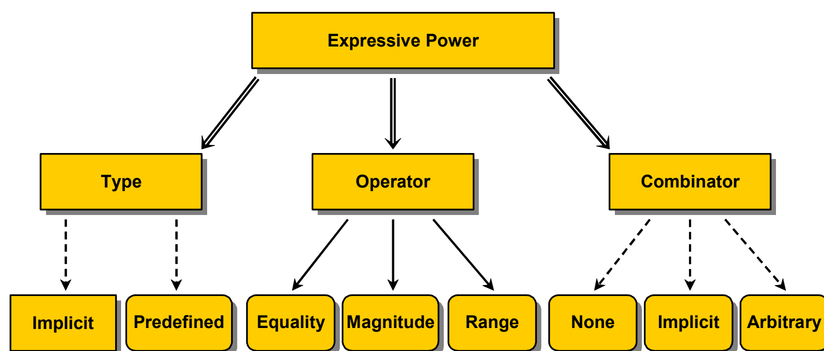


Figure 3.29. Event filter expressive power.

Mobility. Another functional event service property, which is becoming increasingly important with the emergence of technologies for mobile devices and components, is the support for entity mobility. Figure 3.30 summaries the degree of mobility that may be provided by an event service.

Many event services do not support mobility; all entities of such an event system have a **static** location. However, an event system may contain software components representing entities that may move location from one host machine to another thereby assuming the address of the current host machine. The **mobile code** category refers to event services that support entities that can move from one computer to another and subsequently execute at their destination. JEDI supports this feature through its concept of reactive objects. Loke et al. [80, 81] propose an extension to Elvin that enables mobile code, referred to as mobile

agents, to migrate from one host to another in order to perform computations on behalf of mobile multi-agent applications.

The **mobile device** category is used to refer to event services that support portable computing devices, such as notebook computers and handheld devices, which may move location while keeping their addresses, thereby moving the entities they host. Mobile devices may host nomadic and collaborative entities and may be capable of wireless networking. **Nomadic entities** interact through either a fixed network infrastructure or a mobile computing environment to which they connect via nodes acting as access gateway. Characteristically, they may suffer periods of disconnection while moving between points of connectivity. For example, SIENA's mobility support service allows nomadic entities to connect to proxy components using wireless connections based on GPRS technology. These proxy components run on event servers that act as access points and transparently manage (and synchronise) subscriptions and events on behalf of a moving entity. In contrast, **collaborative entities** use a wireless network to interact with other mobile entities that have come together at some common location. Collaborative entities may use ad hoc networks to support communication without the need for a separate infrastructure, thus allowing loosely coupled entities to communicate and collaborate in a spontaneous manner.

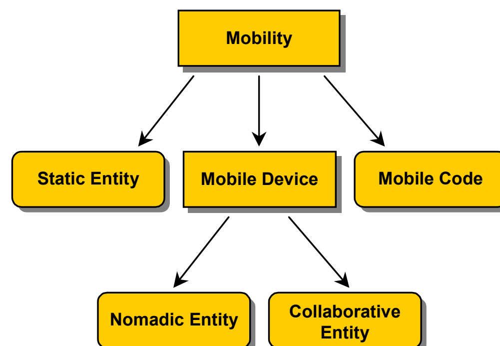


Figure 3.30. Mobility support.

Composite Events. Subscribers may require an event service to recognise the occurrence of a specific pattern of two or more particular events possibly propagated by different producers. Such a combination of event occurrences is called a composite event.

As depicted in Figure 3.31, an event service may **omit** composite events. However, when **supported**, the occurrence of composite events causes the service to notify subscribers accordingly. Subscribers may specify the **number** of the events involved and the **time** window in which the events involved must occur. Exactly **two** or **three or more** events may

be defined along with the time window that may be defined **implicitly** by the event service or **explicitly** by the subscriber application. As part of their work on CEA, Bacon et al. [82] have defined an application-level language for specifying occurrences of event sequences of interest. Monitors then use a combination of event filters to detect composite events that conform to these sequences. Pietzuch et al. [64] propose a general composite event detection framework that is similar to the CEA approach in that it also introduces a high-level specification language for event occurrences of interest. However, this framework has been designed to accompany existing event-based middleware architectures and has adopted the interval timestamp model [83] for handling the clock uncertainties that are intrinsic to distributed systems.

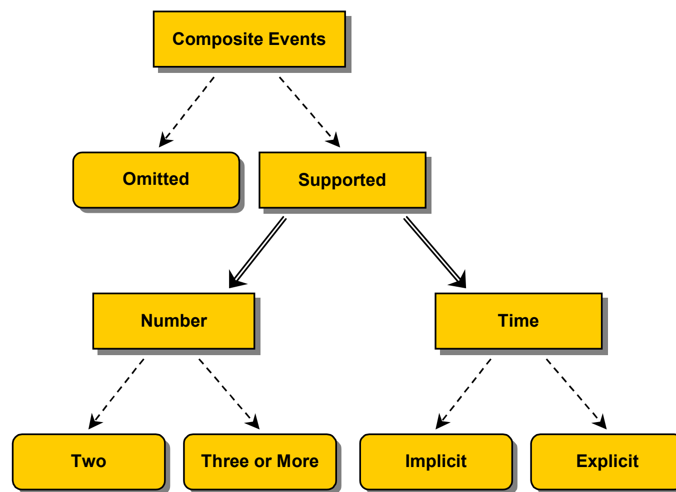


Figure 3.31. Composite events.

Non-functional Features

Quality of Service. The QoS of an event service may be configured according to the requirements of a particular application. Figure 3.32 shows that we divide the QoS supported by an event service into four categories describing the behaviour of an event service when propagating and delivering events.

The **real-time** category explores the guarantees provided by an event service regarding the timely propagation of events. Real-time guarantees can be either best effort, soft or hard. In the **best effort** case, no deadlines can be associated with events. An event service supporting **soft real-time** provides guarantees with a probability that is sufficient to be used for soft real-time deadlines and a **hard real-time** service provides guarantees with a

probability that is sufficiently high to be used for hard real-time deadlines. The CORBA notification service allows deadlines defining earliest and latest delivery time to be assigned to events that are enforced with a probability that is sufficient to be used for soft real-time deadlines. Generally, hard real-time guarantees are difficult to provide as they require predictable communication pattern, ideally utilised in a small-scale environment. This is particularly true for distributed event systems. Distributed event systems are traditionally based on anonymous one-to-many communication patterns that tend to be unpredictable and are likely utilised in systems consisting of a large number of loosely-coupled entities. However, the TAO RT event service, an extension to the CORBA event service that was developed for avionics applications, supports hard real-time guarantees. COSMIC [84] uses event channels as an abstraction for network resources and allows applications to assign timeliness properties to channels. It supports best effort guarantees in the form of non real-time event channels as well as soft and hard real-time guarantees through soft real-time channels and hard real-time channels respectively.

In order to influence the sequence in which events are delivered, a **priority** may be assigned to an individual event. Usually, **no** priority can be assigned and therefore all events have identical priority. An event service that supports **alarm** events allows a single priority to be assigned to certain events. The CORBA notification service provides **multiple** priorities.

The **store occupancy** describes the maximum size of memory required by an event service to operate at any given point in time during its lifetime. This size can be either **implicit** or it may be **configurable** according to the requirements of a particular application. Implicit store occupancy either imposes a fixed maximum memory size or allocates the required memory dynamically whereas configurable store occupancy typically depends on a number of parameters. These parameters may describe the maximum size of the queues that buffer events as well as the maximum number of producers, consumers, and mediators that may be supported by an event service.

The **reliability** category investigates the guarantees provided by an event service regarding the delivery policy of events in the presence of failure. An event service is said to provide **best effort** reliability if no specific delivery guarantees are made. Events may or may not be delivered to subscribers in the presence of failure. An event service that supports **reliable connections** guarantees events being delivered to all correctly functioning subscribers. Upon restart from a failure, connections between producers and subscribers are re-established without re-subscription and event delivery resumes. A **persistent** event service guarantees events being delivered to all subscribers. Upon restart from a failure, connections between producers and subscribers are re-established without re-subscription and persistently

buffered events are retransmitted. The CORBA notification service may support either of these three delivery policies.

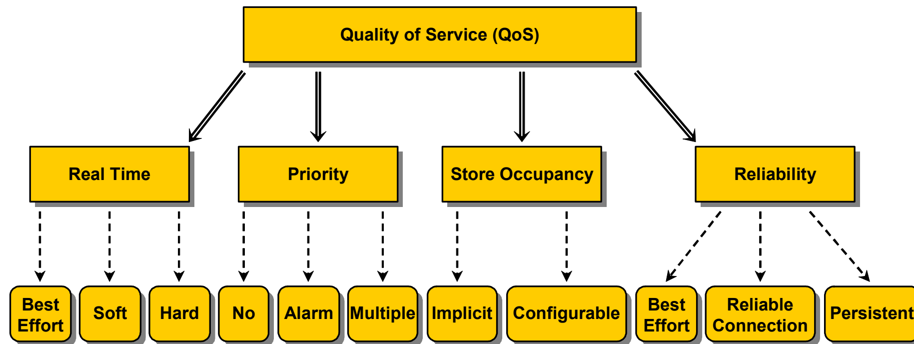


Figure 3.32. Quality of service.

Ordering. An event service delivers events according to a certain order semantics. Figure 3.33 shows that an event service may deliver events in a certain order in a **subset** of the system or **system wide**, i.e., throughout the system. Event services with a system wide ordering strategy employ exactly one delivery order whereas event systems with subset orders associate different ordering strategies with various parts of the system.

Events may be delivered in **any** order. Such unordered events may be received by any subscriber in any order. **FIFO** order refers to a strategy where two events that are raised by the same producer are delivered by consumers with matching subscriptions in the same order they were raised. **Causally** ordered events are delivered in the order they were published while **totally** ordered events are delivered in the same order by all subscribers but not necessarily in the order they were raised [69]. Mechanisms for providing unordered and FIFO order semantics are generally relatively straightforward since they only need to consider events delivered to an individual subscriber. In contrast, enforcing causal and total order semantics requires cooperation between all producers and subscribers involved. Alternatively, events may be delivered according to an associated **priority** or **deadline**. These semantics imply that the delivery of some event can be pre-empted in order to deliver an event that has a higher priority or to deliver an event that has a deadline that is close to expiring. The CORBA notification service supports various semantics for defining event delivery order for a specific event channel, including any, FIFO, priority, and deadline order. This approach allows applications with a single event channel to define a system wide order and applications comprising multiple channels to associate a specific order with each channel. CONCHA and TAO RT are other CORBA-based event services that support

delivery order semantics. CONCHA features totally ordered event delivery and TAO RT CORBA provides a dispatching mechanism for priority-based event delivery.

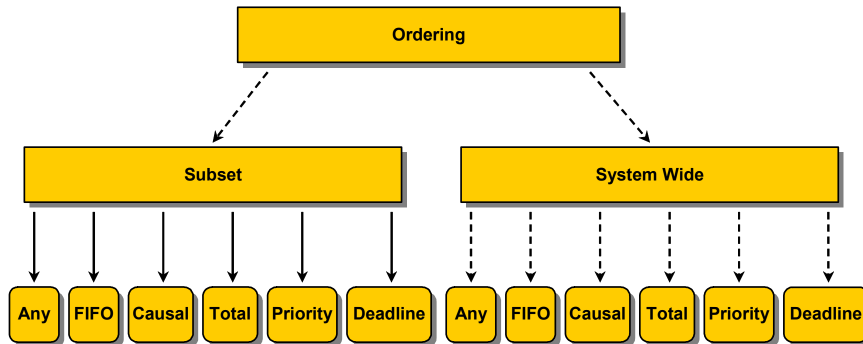


Figure 3.33. Ordering.

Security. Event services may **omit** mechanisms for ensuring security or may support security properties by providing techniques for event message **confidentiality** and for **authentication**.

Event messages that contain sensitive content may be transmitted over a network in an encrypted and therefore confidential form rather than as plain data. This enables producers and consumers to keep event messages secret from third parties. For example, Elvin4 supports a security framework that exploits the Secure Socket Layer (SSL) protocol for managing the security of its message transmissions over the Internet.

Essentially, authentication establishes the identity of specific events and serves as the basis for a mechanism that grants access to certain events. Such an access control mechanism may regulate access privileges for event dissemination, forwarding, and delivery. Access may be granted to an **individual event** or to a **set of events**. Such a set of events may be defined by various means. Access may be granted to events of a specific type, to the events disseminated by a specific producer or group of producers, to the events described by a subscription or by the subscriptions issued by a certain consumer, or to the events handled by a particular mediator. For example, Elvin4's security framework enables servers to authorise access to events using keys, which may be associated with either a connection to a specific entity or an individual event. Wang et al. [85] outline security issues in event services without attempting to present an actual security model. Their work specially focuses on Internet-scale event systems and discusses security paradoxes, such as anonymity vs. authentication, that arise due to the nature of event systems.

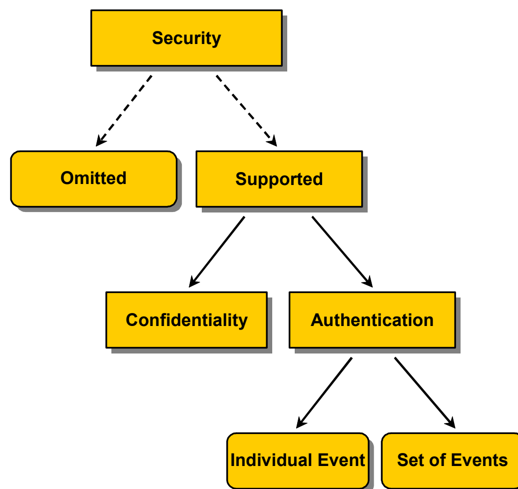


Figure 3.34. Security.

Failure Mode. The failure mode describes the behaviour of an event service in the presence of a single component failing silently. As outlined in Figure 3.35, the failure mode category explores support for the failed component being an entity, a middleware component, or a part of the network.

A silently failed **entity** may be either a **consumer** or a **producer**. A failed consumer does not cause the remainder of the system to suffer. A failed producer causes a partial or a total system failure. A **partial system failure** affects the communication related to some event types that may result in fewer events being propagated. No event communication can take place in case of a **total system failure**. A system consisting of a single producer and a number of consumers fails totally if the sole producer fails silently.

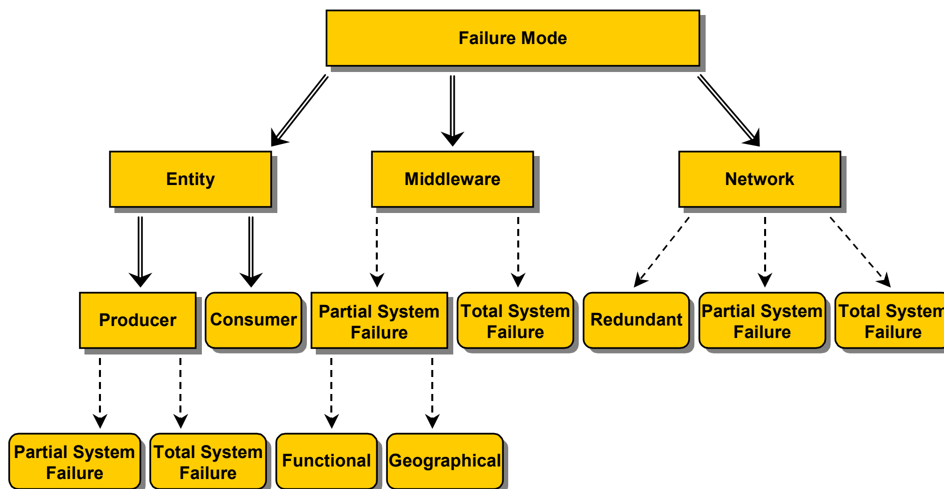


Figure 3.35. Failure mode.

A **middleware** component failing silently causes a partial or a total system failure similar to the effect of a failed producer. A **partial system failure** affects either a **geographical** or a **functional** part of the system. The former disconnects a part of the system from the rest of the system. Event communication may take place within the partitions, but no event communication takes place between the partitions. A geographical partial system failure may be caused by a failing SIENA event server that is part of a hierarchical or an acyclic server topology. The latter stops communication related to a particular event type throughout the system. However, communication related to other event types does not suffer. A functional partial system failure may be caused by a failed event channel in a CORBA event service utilising multiple channels, each managing an specific event type. A failing centralised JEDI event dispatcher causes a total system failure.

A part of the **network** failing silently may be redundant or may cause partial or total system failure. A **redundant** part of the network failing in SIENA utilising a general non-hierarchical server topology may not cause the remainder of the system to suffer.

A **partial system failure** disconnects a part of the system from the rest of the system. Event communication may take place within the partitions, but no event communication takes place between the partitions. SIENA utilising an acyclic non-hierarchical server topology behaves in this manner. A system in which all producers are connected through a single network is susceptible to **total system failure** where no event communication can take place.

3.3 Classification of Event Systems

Table 3.1 illustrates how a taxonomy user may apply the taxonomy to existing event systems by presenting a number of selected event services that have been categorised according to the taxonomy presented in this thesis. These event services have been selected to cover various properties and because sufficiently detailed documentation is available. The commercial CORBA event service and CORBA notification service have been chosen due to their widespread use and in the case of the notification service due to its support of a wide range of non-functional features. SIENA and SECO, which have been designed in academia, have been chosen because of their server topology and their organisation and interaction model respectively.

	CORBA Event Service	**CORBA Notification Service**	**SIENA**	**SECO**
Event Model	Single mediator or multiple, non-functionally equivalent mediators	Single mediator or multiple, non-functionally equivalent mediators	Single or multiple mediators	Implicit
Event Service Organisation	Single or multiple distributed, separated middleware	Single or multiple distributed, separated middleware	Single or multiple distributed, separated middleware	Distributed, collocated middleware
Event Service Interaction Model	Centralised intermediate or partitioned, distributed intermediate	Centralised intermediate or partitioned, distributed intermediate	Centralised intermediate or cooperative, distributed intermediate	No Intermediate, named (uSECO) or implicit (mSECO)
Functional Event Service Features	✓	✓	✓	✓
Event Propagation Model	Sporadic push and pull	Sporadic push and pull	Sporadic push	Sporadic push
Event Type	Generic or Typed	Typed	Typed	Typed
Expressive Power	N/A	Application specific attributes	Application specific attributes	Application specific object
Event Filter	×	✓	✓	✓
Location	N/A	Producer, consumer, and intermediate	Intermediate	Producer and consumer
Definition	N/A	Constraint language	Constraint language	Programming language
Implementation	N/A	String	String	Object
Evaluation	N/A	✓	✓	✓
Mechanism	N/A	Implicit interpreted	Implicit interpreted	Implicit compiled
Time	N/A	Propagation	Propagation	Propagation
Expressive Power	N/A	✓	✓	✓
Type	N/A	Predefined	Predefined	Predefined
Operator	N/A	Range	Range	Range
Combinator	N/A	Arbitrary	Arbitrary	Arbitrary
Mobility	Static	Static	Static and nomadic entity	Static
Composite Events	Omitted	Omitted	Omitted	Omitted

	CORBA Event Service	CORBA Notification Service	SIENA	SECO
Non-Functional Event Service Features	✓	✓	✓	✓
Quality of Service	×	✓	×	✓
Real-time	N/A	Soft	N/A	Best Effort
Priority	N/A	Multiple	N/A	No
Store Occupancy	N/A	Configurable	N/A	Implicit
Reliability	N/A	Best effort, reliable connection or persistent	N/A	Best effort (uSECO) or reliable connection (mSECO)
Ordering	Any	Any, fifo, priority or deadline	Any	Any
Security	×	×	×	×
Failure Mode	✓	✓	✓	✓
Entity	Partial system failure	Partial system failure	Partial system failure	Partial system failure
Middleware	Functional partial system failure or total system failure	Functional partial system failure or total system failure	Geographical partial system failure or total system failure	Results in failed entity
Network	Partial system failure	Partial system failure	Redundant or partial system failure	Partial system failure

Table 3.1. Categorisation of event systems.

3.4 Summary

This chapter presented a taxonomy of distributed event-based programming systems. The taxonomy identifies a set of fundamental properties of event-based programming systems and categorises them according to the event model and event service criteria. The event service is further classified according to its organisation and interaction model, as well as other functional and non-functional features. These properties are then arranged in a hierarchical manner starting from the root of the taxonomy, which defines the relationship between event system, event service and event model. Each of these properties is described in detail and previously surveyed event systems are used as examples.

We consider the event model property to be the most important of the identified properties since it classifies the part of an event system that is exposed to the application programmer. The relative significance of other properties depends on the domain for which a specific event system has been designed.

We have demonstrated how a taxonomy user may apply the taxonomy to existing event systems by categorising a number of selected event services, which have been chosen to cover various properties, according to the taxonomy.

Our taxonomy differs from related work in that it identifies an extensive set of generic event system properties describing various systems dimensions in detail. The taxonomy considers functional and non-functional properties, including mobility, security, and quality of service, and describes the possible options for these properties. As a result, it can be used to classify virtually any distributed event-based programming system regardless of system scale or application domain whereas existing work focuses on providing a framework designed for a specific application area or based on a particular high-level model.

Event systems may evolve together with future advancements in the information technology industry. Such next-generation event systems may support additional, novel properties in order to accommodate new application requirements that may result from these advances. For example, a means for consumers to electronically pay producers for the information they disseminate may arise as an important feature in future event-based systems. Consequently, the taxonomy may need to be extended to support such novel properties. The hierarchical structure on which our taxonomy is based may easily cope with such potential enhancements. Adding novel properties or refining existing properties is straightforward as such changes affect a specific part of the taxonomy only and do not require a reorganisation of the existing hierarchy.

CHAPTER 4: THE STEAM EVENT-BASED MIDDLEWARE FOR COLLABORATIVE APPLICATIONS

STEAM is intended for collaborative applications that include a large number of highly mobile (and stationary) components typically distributed over large geographical areas. This chapter presents the rationale for the design of STEAM and the key concepts that reflect the requirements of collaborative applications. We outline the notification model that has been adopted by STEAM as well as STEAM's approach to filtering event notifications at both the consumer and the producer side according to multiple, functional and non-functional attributes. We then introduce the communications architecture of STEAM before concluding this chapter by presenting STEAM's programming interface and discussing several related issues.

4.1 Proximity-Based Event Notifications

Current approaches to event-based middleware supporting mobility mainly focus on disseminating event notifications to nomadic application components and fail to address the requirements of collaborative applications.

The STEAM event-based middleware provides for the spontaneous nature of collaborative applications by exploiting proximity-based event dissemination. STEAM enables such applications to define geographical scopes in order to bound event dissemination and to associate these scopes with the specific locations at which the corresponding event notifications are relevant.

4.1.1 Event-Based Middleware for Collaborative Applications

Existing research on event-based middleware for mobile computing has essentially concentrated on handling disconnection on behalf of nomadic application components

migrating between points of connectivity. Relatively little work has been done to accommodate collaborative application components that come together at a common location in order to interact.

Accommodating Collaborative Applications

The event-based middleware presented in this thesis supports a style of application in which *collaboration is intrinsic when components are in close proximity*. Components that comprise this style of application are *inherently mobile* and may move together and apart over time. Such components characteristically come together at a certain location and communicate and collaborate and then come together at a different location and collaborate with other components. Event-based middleware accommodating such collaborative applications must therefore enable producers to disseminate event notifications to nearby consumers that are more likely to be interested in event notifications the closer they are located to the publishing producer. The design of the middleware presented in this thesis provides for the following key requirements of collaborative applications:

- Ad hoc wireless communication. The middleware should support an event-based programming model allowing collaborative application components, with significant variations in speed from stationary to highly mobile, to come together at a certain location and then to communicate and collaborate through wireless connections using ad hoc networks.
- Inherently distributed system architecture. The middleware should exclusively use the same physical machine as the components that comprise the collaborative application and not rely on the presence of a designated service infrastructure.
- Event notification filtering precision. The middleware should support a range of event notification filters that may be applied to various attributes of event notifications including subject, content, and context, such as geographical location. Moreover, it should allow a subscriber to combine event notification filters in order to describe the exact subset of event notifications in which it is interested exploiting multiple criteria, such as meaning, time, location, and quality of service.
- System scalability. A system exploiting event-based middleware for collaborative applications should be able to easily cope with a large, dynamically changing population of mobile components distributed over a large geographical area and the resulting dynamic reconfiguration of the connections between the components.

Event-based middleware supporting this collaborative style of application may be used in various areas including indoor and outdoor smart environments, augmented reality, and traffic management. For example, such middleware may be used to support augmented reality games, in which players are interested in the status of game objects or indeed other players, only when they are within close vicinity [86]. An example scenario from the traffic management domain might include a crashed car disseminating an accident notification. Approaching vehicles are interested in receiving these event notifications only when located near the car.

Exploiting Proximity

STEAM provides for unanticipated interactions among collaborative application components by allowing event producers to define geographical scopes, called proximities, that bound the areas in which their event notifications are relevant. Event consumers residing or indeed entering such areas can dynamically discover these proximities and subsequently establish logical connections to the associated producers. The connections between the components residing in a particular proximity are then used by producing components to disseminate their event notifications, thereby allowing consumers to deliver events at the specific location where they are valid. This concept of proximity therefore enables migrating components that have come together at a certain location to spontaneously discover and interact with each other.

Using Wireless Ad Hoc Communications

Several event services for the mobile computing domain have been proposed, including JEDI, Elvin4, Mobile Push, and SIENA. Although there are some variations in these approaches, their respective objectives are essentially to support nomadic application components that use wireless communications to connect to a fixed infrastructure network.

In principal, collaborative application components may also use infrastructure networks. However, due to the spontaneous nature of collaborative applications, they often use ad hoc networks to prevent dependencies on a previously installed service infrastructure. Ad hoc networks are immediately deployable in arbitrary environments and support communication without the need for a separate infrastructure. This enables a changing pool of collaborative components to spontaneously group together anywhere and at any time in order to achieve a common goal. STEAM therefore focuses especially on accommodating the loosely-coupled, collaborative application components that use ad hoc networks.

4.1.2 Programming Model

STEAM provides a programming model that is based on proximity. Essentially, STEAM's programming model enables applications to define geographical scopes and to associate these scopes with certain event notifications. Such geographical scopes represent a natural way for mobile components to identify event notifications of interest and to deal with the geographical dispersion of the system.

Collaborative applications may specify the shapes and the dimensions of their proximities. Each of these proximities is then associated with event notifications of a specific type and mapped to the particular geographical area where these event notifications are relevant. This bounds the areas in which producers disseminate event notifications. Consumers residing inside such proximities can discover event notifications of interest and may subsequently deliver them.

Event Notification Scoping

Although the concept of event notification scoping, which has been introduced in ECO, has also been adopted by Fiege et al. [75], their work uses scopes to provide an abstraction for encapsulation and reusability in heterogeneous environments rather than as a means to identify event notifications of interest for mobile components.

Supporting Mobile Applications

STEAM's programming model provides a fundamentally different approach for supporting collaborative mobile applications compared to the approaches that have been proposed to support nomadic mobile applications. These programming models reflect the fact that their main purpose is to provide mechanisms to (often explicitly) handle disconnection and reconnection, as well as to cache and synchronise event notifications on behalf of disconnected mobile components. For example, both JEDI's and SIENA's respective programming models provide `moveIn` and `moveOut` operations for nomadic components to explicitly connect to and disconnect from a point of connectivity. In contrast, STEAM supports spontaneous interactions between a group of nearby mobile components through the concept of proximity. Proximities allow mobile components to discover event notifications of interest using location information rather than by connecting to a service infrastructure and are therefore suited to support wireless communication using ad hoc networks. Furthermore, STEAM specifically addresses the needs of mobile applications by extending ECO's concept of geographical scopes with a notion of proximity in which areas of interest can be mobile as well as stationary.

4.1.3 Architecture

Architectures of existing event-based middleware for the mobile computing domain reflect the fact that they were designed to use infrastructure networks. STEAM's architecture supports wireless communication utilising the ad hoc network model without the aid of access points or connections to a conventional fixed network.

Middleware Distribution

Event-based middleware traditionally employs logically centralised or intermediate components to implement key middleware features and properties, such as event notification filtering, peer discovery, routing, and non-functional attributes. Such middleware components are typically hosted by physical machines that are part of a designated service infrastructure in order to ensure that they are always accessible to application components.

A similar approach has been used by middleware for nomadic applications since designated middleware components, which typically implement the mechanisms for handling disconnection, can be hosted naturally by parts of the network infrastructure, for example by wireless access points as suggested by Huang et al. [6]. This is also illustrated by SIENA and Elvin4 as both use intermediate proxy components, which are hosted by parts of the service infrastructure, for managing information on behalf of a moving entity. Likewise, JEDI and CEA use intermediate components for event notification dispatching that are part of the network infrastructure.

The architecture of STEAM is inherently distributed and is based on an organisation with distributed collocated middleware [23]. The STEAM middleware is exclusively collocated with application components and depends neither on centralised or intermediate components nor on the presence of a designated service infrastructure.

Middleware Capabilities

STEAM's inherently distributed architecture implies that every STEAM instance offers identical capabilities to its application. In other words, every physical machine hosting STEAM is capable of providing the same service to producers and consumers without accessing remote components. The design of STEAM facilitates this by supporting various distributed mechanisms to provide the desired middleware properties. Consequently, STEAM provides decentralised techniques for peer discovery based on beacons, for routing event notifications from producers to consumers without the aid of access points using multicast groups and a distributed addressing scheme, for enforcing non-functional attributes, such as

event notification delivery order and priority, and for event notification filtering based on combining multiple, producer-side and consumer-side filters.

Event Notification Filtering

Event notifications can not depend on intermediate components applying event notification filters at a central location. STEAM therefore supports a distributed approach to filtering allowing event notifications to be filtered at both the producer side and the consumer side. Supporting event notification filtering on either side enables an application to exploit the advantages of both approaches. Producer-side filtering is efficient as unwanted event notifications are discarded close to their sources thereby limiting the use of communication and computational resources. Consumer-side filtering on the other hand may result in consumers discarding unwanted event notifications after they have been propagated. However, applying filters at the consumer side enables filtering on the context of event notifications, such as the geographical location (of consumers), that simply is not available at the producer side.

STEAM supports a range of event notification filters that may be applied to a variety of event notification attributes, including subject, content, and context, and allows combinations of event notification filters. Significantly, combining event notification filters is beneficial to the precision of filtering allowing a component to define the subset of event notifications in which it is interested using multiple criteria, such as meaning, time, location, and quality of service.

4.1.4 Summary

The design of the STEAM middleware, in particular the design of its programming model and architecture, addresses the requirements of collaborative applications. STEAM's programming model is based on the concept of using proximity to bound the dissemination scope of event notifications. Such proximities represent a natural way for mobile application components to identify event notifications that are of interest at a certain location and to establish ad hoc wireless connections to other components residing in a particular proximity. STEAM's inherently distributed architecture enables wireless communication utilising the ad hoc network model as it depends neither on the aid of access points nor on connections to a conventional fixed network. Furthermore, it avoids designated middleware components that may become communication bottlenecks with increasing system scale. STEAM's middleware capabilities incorporate a range of decentralised mechanisms that have been designed to accommodate changing populations of mobile application components and consequently,

help to improve the scalability of a system. These mechanisms include a distributed approach to event notification filtering based on combining multiple, producer-side and consumer-side filters. Filters may be applied to the subject, content, and context of events and combining them is beneficial to the precision of event notification filtering.

4.2 The STEAM Event Model

The STEAM event model has been designed to disseminate proximity-based event notifications to the components that comprise collaborative applications. As illustrated in Figure 4.1, collaborative applications are often hosted by mobile devices, which may migrate with significant variations in speed from stationary to highly mobile, and communicate through wireless connections using ad hoc networks. Collaborative applications that use STEAM can act either as consumers or as producers of event notifications, or indeed as both.

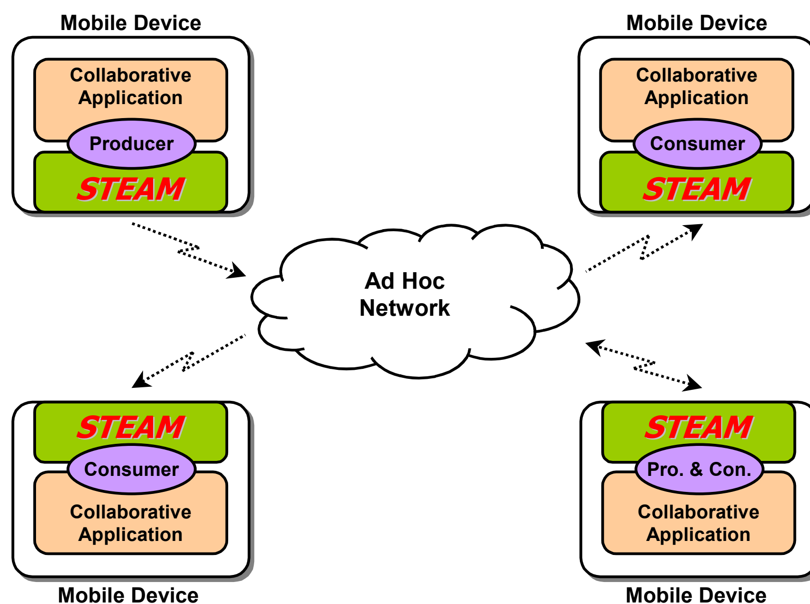


Figure 4.1. STEAM event model.

STEAM implements an implicit event model [23] allowing entities to interact using event types, thereby providing a notion of event-based communication where entities are anonymous to each other but known by the middleware. Producers publish event notifications of a specific type and consumers can subscribe to event notifications of particular types. Producers may publish event notifications of several types and consumers may subscribe to one or more event types.

4.2.1 Supporting Mobility

The STEAM event model supports both stationary and mobile entities and uses proximities for specifying the geographical scopes in which certain event notifications are relevant. Figure 4.2 shows that this notion of proximity is defined firstly by the covered area, which is described as a geometric shape with associated dimensions and a reference point that is relative to this area, and secondly by a naval location. Proximities may be either stationary or mobile. The reference point of a stationary proximity is attached to a naval represented by a fixed point in space whereas the reference point of a mobile proximity is mapped to a moving naval, which is characteristically represented by the location of a specific mobile producer. Hence, a mobile proximity moves with the location of the producer to which it has been attached. For example, a group of vehicles heading in the same direction may cooperate to form a platoon in order to reduce their consumption of fuel. These vehicles might interact using a mobile proximity that has been defined by the leading vehicle. Such a proximity might be attached to a naval represented by the position of the leader thereby moving with its location.

Proximity = {Area(Shape, Dimensions, Reference Point), Naval}

Figure 4.2. Proximity definition.

Stationary Scopes

Figure 4.3 depicts an application scenario that involves a stationary proximity. P_C represents a producer acting on behalf of a crashed car that is blocking the road. P_C disseminates an accident notification to approaching vehicles to prevent them from driving into the obstacle. To facilitate this scenario, P_C defines a circular shaped proximity and attaches the reference point at the centre of this area to the naval defined by the location of the accident site. P_C thereby attaches this stationary proximity to the fixed location where the accident occurred and bounds the scope within which accident events are disseminated. Approaching vehicles discover this stationary proximity and can receive accident events once they reside inside the proximity.

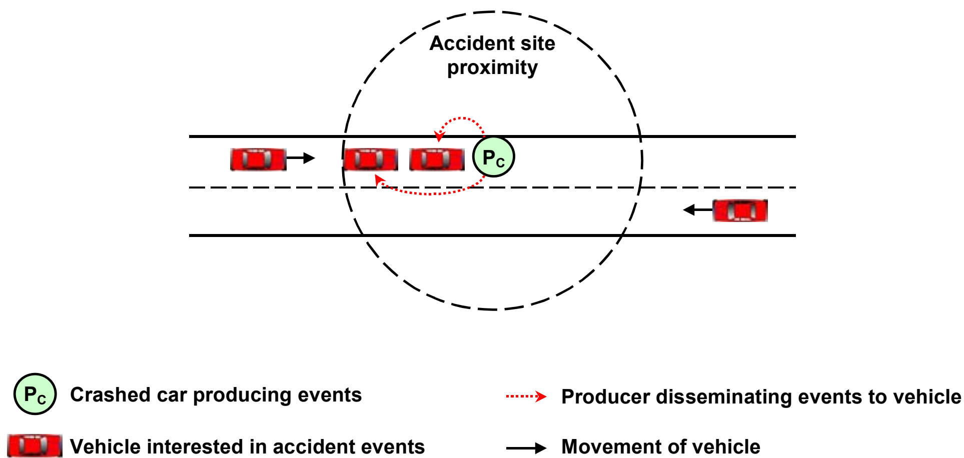


Figure 4.3. Stationary proximity.

Mobile Scopes

Figure 4.4 illustrates a further application scenario exploiting a mobile proximity. P_A represents an ambulance rushing to an accident site using a mobile proximity to inform nearby vehicles of its location in order for them to pull over and yield the right of way.

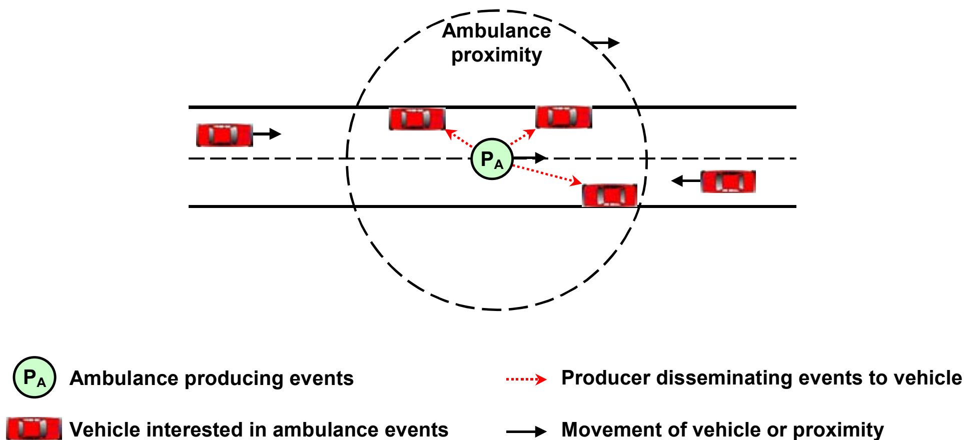


Figure 4.4. Mobile proximity.

Prior to disseminating events, P_A defines a circular shaped proximity and attaches the reference point at the centre of this area to the naval defined by the actual position of the ambulance. This causes this mobile proximity to migrate with the ambulance. Vehicles residing inside the proximity can receive ambulance events and temporarily suspend their journeys while the ambulance is passing them. These vehicles may resume their travels once

the ambulance and its proximity have left. Vehicles travelling outside the proximity will not receive these events and may therefore continue their respective journeys regardless.

4.2.2 Proximities

The STEAM event model combines the concepts of exploiting proximity to bound the range within which event notifications are valid and of using event types to specify the kind of event notifications producers intend to publish and in which consumers are interested. Producers specify and then announce the type of event notification they intend to raise together with the proximity within which events of this type are to be disseminated. Such an announcement associates a specific event type with a certain proximity and implicitly bounds event propagation. Announcements allow consumers to discover event types of interest as well as the proximities that have been associated with them. Consumers can then receive event notifications if (and only if) they reside inside a proximity in which events of an announced type are raised.

Defining Proximities

In principal, either a consuming or a producing entity may define a proximity in which events of a specific type are relevant. Consumers might wish to define proximities that describe their interest in events published in certain areas depending on their (current) activities. For example, a migrating consumer might define its scope of interest according to its actual travel speed. Producers, on the other hand, might wish to define proximities describing the scopes inside which their event notifications are valid. However, we believe that in many application scenarios it is the producer that would define proximities, thereby allowing an application to bound the scope within which specific events are disseminated. A producer may assess its local conditions, which likely apply to all consumers within its vicinity, and may determine, based on application requirements and these circumstances, the specific validity of its events and consequently define an appropriate proximity. For example, a traffic light propagating its status to approaching vehicles defines its proximity based on the location of the next traffic light and on the local speed limit. Nevertheless, we admit the possibility of applications in which consumers might wish to determine their proximities. A vehicle exceeding the local speed limit, for example a police car on a call, may require a larger scope for receiving a traffic light’s status compared to “ordinary” vehicles travelling within a given speed limit.

Shape and Dimension

Producers are free to define the geometric shape and the dimensions of a proximity according to the requirements of the application. A proximity may be of arbitrary complex shape and its dimensions may be determined independently of the producer's physical radio transmission range as the underlying transport mechanism uses a multi-hop protocol to route event messages from producer to consumer.

Complexity vs. Efficiency

The complexity of the shape of a proximity may vary from simple to complex. Such shapes may range from two dimensional circles, squares, or polygons to three dimensional spheres, cubes, or cylinders. However, the more complex the shape of a proximity is the more demanding it will be to determine whether an entity resides within its boundaries. Hence, applications have to consider the trade off between using a complex shape that describes an area of interest exactly and a simpler shape that approximates an area for which membership can be determined more efficiently.

Location

Proximities may be defined as nested and overlapping areas. Nesting allows a large proximity to contain a set of smaller proximities subdividing the large area. Figure 4.5 depicts two overlapping proximities of different shape and illustrates that multiple consuming and producing entities may reside inside a certain proximity. These proximities have been associated with events of type A and type B respectively. Consequently, consumers handling these types receive events if they reside inside the appropriate proximity. Note that entities located inside these areas handling other event types will not affect the propagation of these events, assuming sufficient communication and computational resources are available. Furthermore, since proximities may be mobile, their relationship, in terms of nesting and overlapping, may change over time.

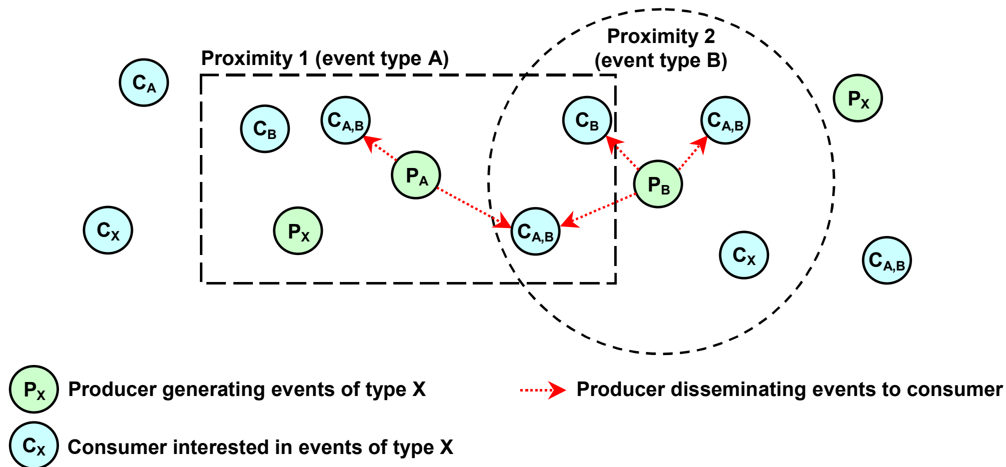


Figure 4.5. Disseminating event notifications using overlapping proximities.

An example of overlapping stationary proximities might include a car disseminating an accident notification within the vicinity of a traffic light that propagates its status to approaching vehicles. An ambulance propagating a warning while travelling through the scope of the traffic light might serve as an example of a mobile proximity temporarily overlapping with a stationary proximity.

4.2.3 Event Types

The types of STEAM events provide an explicit and expressive data structure into which event data can be mapped. Producers must define the event types that appropriately describe their event data prior to disseminating event notifications of these types. Consumers must subscribe to event types in order to have the middleware deliver subsequent event notifications to them if they are located inside *any* proximity where event notifications of this type are raised until they unsubscribe.

Subscriptions are Persistent

A consumer may move from one proximity to another without re-issuing its subscriptions. Thus, subscriptions are persistent and will be applied transparently by the middleware every time a subscriber enters a new proximity. This implies that a subscription to a specific event type applies to all proximities handling these events even though the subscriber may only receive a subset of these events at any time. A single subscription may result in events of a particular event type raised by different producers in multiple proximities being delivered.

Hence, the set of events received by a subscriber at a given time depends on its migration as well as on the movements of producers and proximities.

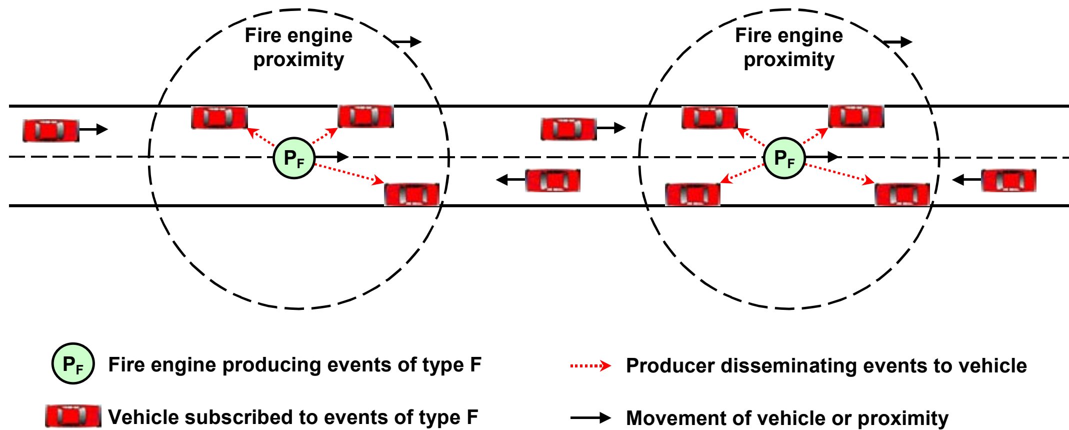


Figure 4.6. Moving between proximities.

The example of Figure 4.6 shows two fire engines disseminating events of the same type within their respective proximities. These events advise vehicles in a fire engine's vicinity to stop and yield the right of way. A vehicle passing through either or both of these proximities will receive these events without having to re-issue its subscription.

Subject-Based vs. Content-Based Event Notifications

Event-based communication models often support a subscription mechanism that is based either on the subject *or* the content of an event notification. The STEAM event model supports an approach that enables consumers to subscribe to the subject *and* the content of an event, thereby allowing applications to combine the ease of use of the subject-based approach with the expressiveness of content-based mechanisms.

Defining Event Types in STEAM

Applications define event types to specify the functional and non-functional attributes of the events they intend to disseminate. Figure 4.7 illustrates that a STEAM event type consists of subject and content representing its functional attributes, as well as of a self-describing attribute list representing its non-functional attributes. The subject defines the name of a specific event type and the content defines the names and types of a set of associated parameters. Non-functional attributes are self-describing in that they outline their respective objective and provide a value to quantify this objective. These attributes describe the non-

functional properties of events, such as their context, dissemination semantics, and their quality of service requirements.

content_name_type = {(par_name$_1$, par_type$_1$), (par_name$_2$, par_type$_2$), ..}
STEAM event type = {subject, content_name_type, attribute$_1$, attribute$_2$, ..}

content_value = {par_value$_1$, par_value$_2$, ..}
STEAM event instance = {subject, content_value, attribute$_1$, attribute$_2$, ..}

Figure 4.7. STEAM event type and instance definition.

Figure 4.7 also outlines that a STEAM event instance is defined in a similar manner by specifying a subject, values for the corresponding content parameters, and an attribute list that applies to a particular event instance rather than to an event type.

Definition Language

Producers and consumers must use a common vocabulary defined by the application to agree on the name of an event type. As illustrated by CEA, event models may define a specific language for defining event types similar to CORBA's interface definition language. Such languages can be used to pre-process the event types defined by a system and to verify type compatibility.

Functional Attribute	Value Type
Subject	Subject String
Content Name	Parameter Name String
Content Type	Parameter Type String, Integer, Floating Point, Location Coordinates, Time

Table 4.1. Defining functional attributes.

STEAM exploits an event type system based on a meta definition language that does not require pre-processing and thus better suits the dynamic nature of collaborative applications. This language enables applications to define event types using the set of functional and non-functional attributes summarised in Table 4.1 and Table 4.2 respectively. Subjects and content names can be defined using character strings while content types can be selected from a predefined set of parameter types. The available non-functional attributes define the

quality of service requirements of event types as well as the routing requirements and the location and time context of event notifications. Associating a proximity with an event type results in defining the scope to which a set of functional *and* non-functional attributes applies. Hence, this bounds the dissemination scope of event notifications and specifies a quality of service zone that describes the non-functional requirements of certain events.

Non-Functional Attribute	Value Type / Range
Event Notification Class	NRT, SRT, HRT
SRT/HRT Deadline	Time
NRT Expiration Time	Time
NRT Delivery Order	Unordered, FIFO, Causal, Total
Persistence Level	Integer
Message Type	Single-Hop, Multi-Hop
Producer Location	Location Coordinates
Forwarder Location	Location Coordinates
Publication Time	Time

Table 4.2. Defining non-functional attributes.

Quality of Service. The quality of service requirements of an event type can be defined using a combination of attributes that classify the timeliness requirements of event notifications as either hard, soft, or best effort. Hence, an event type and its events can be classified as Hard Real Time (HRT), Soft Real Time (SRT), or Non Real Time (NRT). HRT and SRT classes require an additional attribute defining the actual delivery deadline of event notifications while NRT event types allow applications to define optional attributes describing a time for discarding expired event notifications and an explicit delivery order. HRT and SRT event types do not define an explicit delivery order as this would contradict the implicit delivery order described by delivery deadlines. Applications select appropriate deadlines for their events according to their requirements and depending on the dimensions of their proximities. The class of an event type also implies its reliability guarantees. HRT and SRT events are delivered according to an exactly once semantics whereas NRT event delivery is best effort. The mechanisms for enforcing event classes are likely to be affected by the dynamic nature of the mobile computing environment for which STEAM has been designed. Attributes that impose the quality of a service depend on the resources made available from the underlying network, which may change with entity and indeed proximity migration. A technique for achieving timeliness and reliability for real-time event-based communication in ad hoc wireless networks has been proposed by Hughes and Cahill [87]. Their conceptual model is the first to directly address the issue of achieving timeliness and reliability in dynamic

networks and essentially relies on predictive techniques to alleviate the impediments to real-time event-based communication that are characteristic of mobile ad hoc environments. This model essentially allows the non-functional requirements of an event type to be mapped to the quality of service zone that is defined by the associated proximity. A proactive technique for reserving the required network resources based on predictions on the future behavior of mobile entities and indeed proximities is used to route messages from a producer to consumers with a high probability. Providing strong guarantees in large scale, mobile computing systems is usually more expensive than supporting a weaker quality of service while weaker guarantees often support higher performance. Applications need to consider this trade off when defining the classes of their event types.

Delivery semantics. The semantics of the delivery order of NRT event types can be characterised as either unordered, FIFO ordered, causally ordered, or totally ordered [69]. Unordered event notifications, which is the default ordering semantics, may be received by any subscriber in any order. Two FIFO ordered event notifications that are raised by the same producer are delivered by consumers with matching subscriptions in the same order they were raised. Causally ordered events are delivered in the order they were published while totally ordered events are delivered in the same order by all subscribers but not necessarily in the order they were raised. Causal and total ordering therefore affect event delivery when consumers receive event notifications from multiple producers, for example when consumers are members of overlapping or nested proximities. Members of such groups are required to cooperate in order to deliver their events in the correct order.

Persistence. The attribute describing the persistence level of an event type defines the maximum number of event notifications that producers cache on behalf of temporarily unavailable consumers. Maintaining a large number of cached event notifications may require substantial computational resources, which may be scarce on certain mobile devices, and thus, the default persistence level is zero indicating that no events are stored. In many application scenarios it often suffices to accurately describe the state of a producer by buffering the most recently disseminated event. For example, a traffic light disseminating its status can describe its current state by storing the latest event as previous light changes have become obsolete. Similarly, an ambulance propagating approach warnings might only store the event comprising its latest location.

Routing Strategy. The message type attribute allows applications to select an appropriate routing strategy. Depending on the dimensions of a proximity and the radio range of the available wireless transmitter event notifications can be disseminated using either single-hop or multi-hop messages.

Context. Location-based and time-based attributes describe the context of individual event notifications. Such context information can be exploited at application level and geographical location context represents the key enabling information required for supporting the concept of using proximity to bound event dissemination.

Conflicting Attributes

These attributes are not all orthogonal to each other. Attributes defined by different event types might contradict each other in the presence of overlapping and nested proximities for instance, when defining distinct quality of service requirements or ordering semantics. Moreover, attributes may compete for shared communication and computational resources. In cases where attributes describing quality of service requirements or ordering semantics conflict, the attributes defining the stronger semantics take precedence. Figure 4.8 illustrates the dependencies between such non-functional attributes outlining their precedence from left to right. For example, HRT event type classes are prioritised over SRT classes, which precede NRT event types.

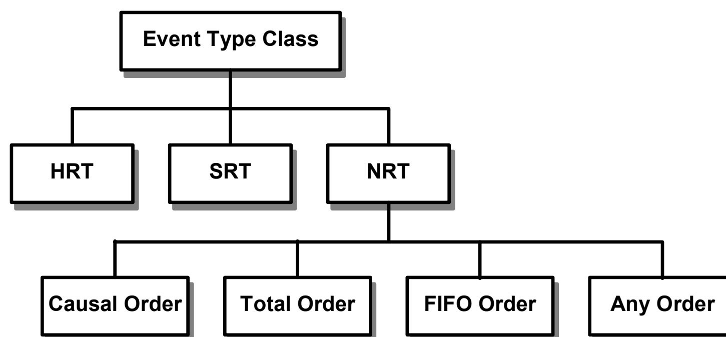


Figure 4.8. Dependencies between non-functional attributes.

Type Safety

Explicitly defining the type of event notifications enables the middleware to use this information when marshalling event notifications for transmission. In particular, the provided name and type information is essential for unmarshalling serialised event notifications. This approach also enables middleware to provide some notion of type safety for its event notifications without relying on a pre-processor. Type information defined using the meta language can be used to verify that the structure of individual event notifications conforms to their respective event types. Such type checks can be performed at run time, for example

when event notifications are instantiated by producers. The structure of event notifications is verified to be consistent with its event type according to the following rules:

- The subject of the event notification is identical to the subject defined by the event type.
- The number of the values describing functional attributes is equal to the number of event type parameters and the type of each of these values conforms to the type of its parameter.
- The number of the values describing non-functional attributes equals the number of the subset of non-functional event type attributes that require values from the application and these values match the type of their respective attribute.

Event notifications that do not match these rules are returned to the producer without being disseminated. Verifying the type of event notifications prior to their propagation causes additional computation overhead for producers but allows consumers to receive events that are consistent with their event types (assuming that they have been transmitted correctly). This prevents consumers from having to validate their respective copies of event notifications at the application level. However, this does not prevent applications from defining conflicting event types with identical subject and different sets of functional and non-functional attributes. Applications must use a common policy when defining the names of event types in order to avert such type inconsistencies.

Attributes and Filters

Many of the non-functional attributes that express the semantics of event types are often applied at a different (lower) middleware layer than event filters. As a result, system resources might be used to enforce a certain semantics, for example in terms of timeliness or ordering, on event notifications that may subsequently be discarded by a filter.

Classification of Attributes

STEAM event types allow applications to associate attributes either to an event type or to a specific event instance. Such attributes apply to either a group of event notifications, i.e., to event notifications of the same type, or to an individual event notification. As summarised in Table 4.3, attributes can be categorised according to the level to which they apply and the means by which they are specified. The values of attributes that are type specific, such as the subject, the quality of service class, and the delivery semantics of event notifications, are specified by applications when defining their event types. Attributes that apply to specific

event notification instances can be further subdivided depending on whether their values are provided explicitly or implicitly. The values of attributes describing delivery deadlines and expiration times must be explicitly specified by the application together with subject and content when raising a particular event notification. The subject identifies the event type of an individual event notification and hence, must be provided by both event type and event notification. The values of attributes that describe the context of specific event notifications can be provided implicitly by the middleware. For example, actual values describing the geographical location of the producer or the time at which an event notification is published are assigned by STEAM, thereby simplifying the programming model exposed to an application.

Category	Applied To	Specified By	Attribute
Event type	Event type	Application	Subject, Event Notification Class, NRT Delivery Order, Persistence Level, Message Type
Explicit event instance	Event instance	Application	Subject, Content, SRT/HRT Deadline, NRT Expiration Time
Implicit event instance	Event instance	Middleware	Producer Location, Forwarder Location, Publication Time

Table 4.3. Classification of attributes.

STEAM Event Type Example

The example of Figure 4.9 shows an event type and an event instance of a traffic light publishing its status, which serves to further illustrate the concept of using event types (and attributes) in STEAM. The event type contains a subject “Traffic Light” and a set of two content parameters describing the light status and the occurrence of the light change as name and type pairs. The event type also comprises a set of attributes describing its non-functional requirements. The first three non-functional attributes are event type specific and define the notification class, the delivery semantics, and the type of message used for disseminating event notifications. The remaining non-functional attribute relates to a particular event instance and describes its producer location.

```
TL_content                 = {("Light Status", String), ("Occurrence", Time)}
Traffic Light event type   = {"Traffic Light", TL_content,
                                Event Notification Class = NRT, Delivery Order = FIFO,
                                Message Type = Single-Hop, Producer Location}

TL_content                    = {"Green", Mon Aug 25 12:25:46 2003}
Traffic Light event instance  = {"Traffic Light", TL_content,
                                   Producer Location = (5320.0N, 615.0W)}
```

Figure 4.9. Traffic light event type and event instance example.

An event instance raised by the traffic light consequently contains the subject and parameter values that correspond to the previously defined parameter types. The event instance specific non-functional attribute provides coordinates describing the location of the traffic light. Note that the value of this location attribute is implicitly provided by the event service.

4.3 Event Notification Filtering in STEAM

An event system may consist of a potentially large number of producers, all of which produce events that may contain different information. As a result, the number of events propagated in an event-based system may be very large. However, a particular consumer may only be interested in a subset of the events propagated in the system. Event filters provide a means to control the propagation of events. Ideally, filters enable a particular consumer to receive only the exact set of events in which it is interested. Events are matched against the filters and are only delivered to consumers that are interested in them, i.e., for which the matching produced a positive result.

4.3.1 Exploiting Distributed Event Notification Filters

The STEAM event model supports a distributed approach to filtering that allows an application to define event notification filters at both the producer and the consumer side. Filters may be matched explicitly at either the producer or the consumer side or may be evaluated implicitly. Furthermore, they may be applied to a range of functional and non-functional attributes associated with an event notification, regardless of whether these attributes have been specified at event type or at event instance level.

Filtering on Functional Attributes

Filters can be applied to the functional attributes of an event notification, namely to its subject and content. A consumer can therefore specify the type of the event notifications in which it is interested and may define content predicates for selecting a subset of these event notifications.

Filtering on Non-Functional Attributes

In addition to supporting event notification filtering on functional attributes, the STEAM event model supports filtering on the non-functional attributes that can be associated with event notifications. Such filters may be applied to a variety of non-functional attributes ranging from context, such as geographical location, to temporal validity and the quality of service available from the network. Filtering on geographical location serves as the basis for proximity-based event notification dissemination. Applications may associate a proximity with a certain event type, which can then act as an implicit event filter using location information to determine whether or not to deliver an event notification to a particular consumer. Attributes describing the temporal validity of event notifications can be used to implicitly filter expired events, whereas quality of service attributes may be used to filter event notifications according to their timeliness requirements. Such quality of service attributes may filter some events in order to enable the timely delivery of other event notifications according to their precedence.

Filtering Precision

The STEAM event model allows an application to specify multiple event notification filters, each of which may apply to a different attribute of a specific event notification. In other words, several event notification filters may be combined and event notifications are only delivered to consumers for which all filters match. Combining filters is beneficial to the precision of filtering allowing a subscriber to define the subset of event notifications in which it is interested using multiple criteria, including not only the meaning of an event notification, but also criteria such as time and geographical location.

Distributing the Filtering Load

Event notification filtering at both the consumer and the producer side implies that a relatively small number of filters is applied on a specific physical machine compared to traditional approaches in which an arbitrarily large number of filters is evaluated sequentially on a single machine hosting a mediator or a producer. Distributed event notification filtering allows

mobile devices to concurrently evaluate the filters that apply to a particular event. The computational load of filter matching can therefore be distributed between several mobile devices, which typically have limited computational resources.

Supporting Mobility

Event notification filtering based on subject and geographical context enhances the ability of a system to accommodate the dynamically changing population of mobile entities by dividing the system into bounded geographical scopes. A moving entity affects only the configuration of the scope inside which it resides. An entity entering a scope causes other entities in the same area to reconfigure, i.e., to update routing and subscription information, without affecting entities residing outside the area. Consequently, exploiting proximity bounds the propagation range of event notifications *and* of event notification filters. In particular, such proximities limit the forwarding of event notifications and filters to a confined geographical area.

4.3.2 Applying Distributed Event Notification Filters

The STEAM event model allows applications to define event notification filters that can be applied to the subject, the content, and the geographical location of individual event notifications.

Essential Filters

STEAM supports three essential event notification filters, namely subject, content, and proximity filters. These filters may be combined and a particular event is only delivered to a consumer if all filters match. Subject filters match the subject of events allowing a consumer to specify the event type in which it is interested. Content filters contain a filter expression that can be matched against the values of the parameters of an event. Content filters are specified using filter expressions describing the constraints of a specific consumer. These filter expressions can contain equality, magnitude, and range operators as well as ordering relations. They may include variable, consumer local information, such as the consumer’s geographical location. Proximity filters are the location filters that define the geographical scope within which event notifications of a specific event type are relevant. These three filters allow consumers to specify the set of event notifications in which they are interested using the meaning and the geographical location criteria.

Matching Filters

Producers and consumers implicitly specify their subject filters by subscribing to or announcing event types. In addition, consumers may define content filters when subscribing to event types. Similarly, producers may define proximity filters when announcing their event types. Consumers obtain the proximity filter of a specific scope when discovering a proximity associated with one of their event types. Figure 4.10 illustrates that both producers and consumers may implicitly apply subject and proximity filter pairs to determine whether their current location is within the geographical scope of a particular event type. As a result, events are only delivered to a consumer if both subject filter and proximity filter match. The consumer subsequently matches a received event notification against its content filter to determine whether or not to deliver it to the application.

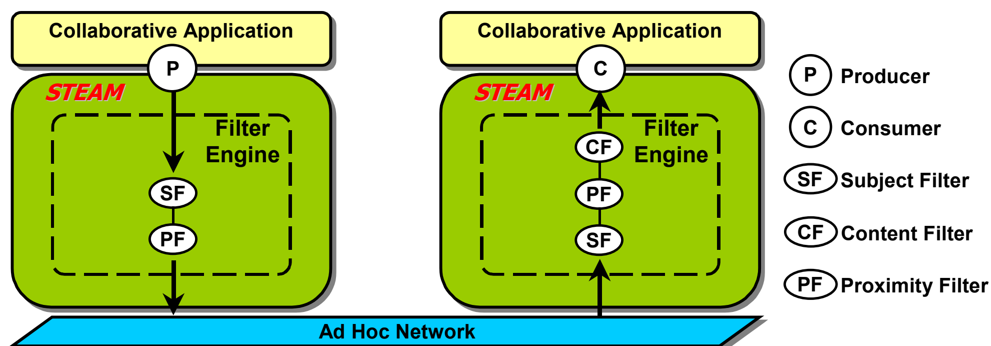


Figure 4.10. Matching distributed event notification filters.

Applying distributed event filters to functional attributes allows a system to combine the advantages of both subject filters and content filters. Subject filters allow efficient event matching, as a simple, subject-based table lookup algorithm can be used to evaluate them. Content filters are expressive filters that can be matched against the parameter values of event notifications. Applying content filters on the consumer side prevents consumers from having to pass content filters to producers when subscribing. This implies that content filters are not forwarded to producers when a consumer changes its location from one scope to another and results in a simple, scalable approach to maintaining subscriptions and content filters.

Maintaining Filters

The traditional approach of applying event filters on the producer side or on intermediate middleware components results in matching an arbitrarily large number of filters on a single

machine. The number of filters depends on the number of subscribers as every one of an arbitrarily large number of subscribers may provide a different filter. Hence, it will be difficult to implement an efficient algorithm for retrieving and matching filters. In contrast, distributed event filtering causes a relatively small, subscriber independent number of filters being matched on each individual physical machine. Both content filter and proximity filter can be retrieved using the event subject as the key for a simple table lookup algorithm, which results in a constant retrieval time. We expect the number of filters to be retrieved on a specific machine to be relatively small. The retrieved proximity filters represent the scopes inside which an entity is currently located while the retrieved content filters represent the subscriptions to a particular subject on a single machine. Most likely, there will be a single proximity filter and a single content filter to be matched for each even type.

4.3.3 Defining Distributed Event Notification Filters

Producers may define subject filters and proximity filters when announcing the event types they intend to disseminate while consumers define their subject filters and content filters when subscribing to these event types. The remainder of this section outlines the concept of entities defining their respective event notification filters by presenting event notification filter examples that might be applied to event notifications describing the status of a “Flag” object in an augmented reality game. Such an object may represent a bonus of some kind that can be collected by players participating in the game. The content of event notifications published by “Flag” objects, which is shown in Figure 4.11, may describe name and type of the parameters that refer to the type and the location of the flag as well as to the bonus value it currently represents.

```
subject = {“Game Flag”}
content = {(“Flag Type”, String), (“Flag Position”, Location), (“Flag Value”, Integer)}
```

Figure 4.11. Example of the subject and content of an event notification.

Subject Filter

Both producers and consumers define their subject filters simply by stating the subject of the event type of interest. As illustrated in Figure 4.12, the subject filter in our example refers to the event type that describes the status of the “Flag” object.

```
subjectFilter = {"Game Flag"}
```

Figure 4.12. Defining a subject filter.

Content Filter

Content filters may consist of an arbitrary number of self-describing filter terms. As shown in Figure 4.13, each term specifies the parameter to which the term applies along with an operator and a value representing the second operand. These filter terms are matched against the relevant parameters of an event notification either in a conjunctive or a disjunctive manner, thus defining whether *all* or *at least one* of the terms that comprise a filter must be true for the filter to match.

Filter Term = {Parameter Name, Operator, Value}

Figure 4.13. Filter term definition.

The example in Figure 4.14 depicts a conjunctive content filter comprising four filter terms. The first of these terms matches a specific type of flag while the combination of the third term and the fourth term matches a certain value range. The second term demonstrates that content filters may contain filter expressions that can be applied to parameter values describing a geographical location. Such a filter expression implicitly uses the actual location of the consumer receiving the event notification when evaluating the filter expression. Location-aware applications may use such a filter expression in order to determine whether the consumer that defines the content filter is moving towards or indeed away from the location stated by an event notification parameter.

```
contentFilter = conjunctive{("Flag Type" == "Blue"),
                            ("Flag Position" DISTANCE_DECREASES),
                            ("Flag Value" >= 50),
                            ("Flag Value" <= 100)}
```

Figure 4.14. Defining a conjunctive content filter.

Proximity Filter

Proximity filters specify the shape, dimensions, and reference point of a geographical area as well as the location of the naval to which the shape is mapped. The naval may be either stationary or mobile, thus referring to a fixed point in space or to the position of a moving entity. The example in Figure 4.15 shows a filter for a circular shaped proximity whose reference point is implicitly defined as the centre of the shape. The naval location of this stationary proximity is described using absolute latitude and longitude coordinates. Event notification filters defining mobile proximities use a reference to the service providing the actual location of a mobile device instead, thereby mapping the proximity shape to a moving naval.

```
shape            = circle(radius)
navalLocation    = location(latitude, longitude)
proximityFilter  = stationary{shape, navalLocation}
```

Figure 4.15. Defining a stationary proximity filter.

4.4 Communications Architecture

The design of the STEAM communications architecture is motivated by our approach of bounding the scope within which certain information is valid and by the characteristics of the underlying wireless ad hoc network. We employ a transport mechanism that combines the concepts of proximity and group communication and use a multicast protocol to route messages between producers and consumers.

4.4.1 Exploiting Proximity Groups

Classical group communication [69, 88, 89] provides a one-to-many or many-to-many communication pattern typically based on a reliable multicast protocol that allows a member of a group to send messages to all members of that group. This communication pattern can be used by producers to propagate event notifications to a group of consumers exploiting the delivery semantics associated with the group. Group communication has therefore been recognised as a natural means to support event-based communication models [7, 90].

An extension to classical process groups, called proximity groups [91, 92], has been identified as a useful communication paradigm for mobile applications, especially those using wireless local area networks [7, 91]. Proximity groups allow potentially mobile application components to join a proximity group of interest and subsequently interact with its members once they are within the same geographical area. In contrast, existing research in this area has focussed on mechanisms in which membership is solely (and implicitly) based on the position of application components. Components using routing protocols for group communication based on geocast [93] automatically become a member of a geocast group once located in the associated geographical area. Similarly, the communication groups for ad hoc networks proposed by Roman et al. [94] rely solely on location information to determine whether an application component is admitted to or eliminated from a group.

Functional and Geographical Aspects

Significantly, this notion of proximity groups defines membership by both functional and geographical aspects. The functional aspect, i.e., the name of the group, represents the common interest of group members based on the information that is propagated among them. The geographical aspect, i.e., the geographical area, outlines the scope within which the information is valid. In order to apply for proximity group membership, an application component must firstly be interested in the group and secondly be located in the geographical area that corresponds to the group. In contrast, classical group communication defines groups solely by their functional aspect.

STEAM exploits proximity-based group communication as the underlying means for entities to interact. Application components must therefore identify both the functional and the geographical aspect that specifies a proximity group when applying for group membership [91, 92]. The functional aspect represents the common interest of producers and consumers based on the type of information that is propagated among them and the geographical aspect outlines the scope within which the information is valid, i.e., the area within which the corresponding event notifications are propagated. Hence, STEAM maps the subject of an event type to the functional aspect and the associated proximity to the geographical aspect of proximity groups.

Absolute and Relative Proximity Groups

A proximity group can be distinguished as either absolute or relative. The geographical area associated with an absolute proximity group is geographically fixed whereas the geographical area associated with a relative proximity group is relative to a moving point in space, most

likely one of the proximity group's mobile members. This notion of absolute and relative proximity groups is orthogonal to the functional and geographical aspect of a group and serves as the basis for the concept of exploiting stationary and mobile scopes on which STEAM's programming model is based.

Message Delivery and Membership Management

Similar to traditional group communication, proximity-based group communication provides a means for a group member to disseminate messages to the other members of the proximity group. Proximity groups include a form of membership management that enables them to identify the sets of members to which individual messages are sent. Messages are delivered according to a certain group specific semantics for example, in terms of reliability, timeliness, and ordering. Message delivery is synchronised and as a result, proximity groups buffer messages until their applications are ready to process them.

4.4.2 Locating Proximity Groups

The STEAM event model comprises a distributed mechanism for locating entities that wish to interact rather than relying on a traditional, centralised approach based on exploiting a naming service. This discovery mechanism, which is called the proximity discovery service, is an integral part of STEAM and consequently, runs on every physical machine that hosts a STEAM instance regardless of whether the local entities act either as producers or as consumers, or indeed as both.

Announcing Proximity Groups

The proximity discovery service uses beacons to periodically announce relevant proximities (and the associated event types) on behalf of its producers. The discovery service announces the event type and proximity pairs that have been defined by its producers within the scope of the respective proximity. This implies that the location at which these announcements are disseminated can change when the device hosting a proximity discovery service migrates and that the set of adjacent devices is likely to change as well. Whether or not other stationary or mobile devices forward such announcements for them to reach a larger number of entities depends on their dissemination policy. Such a policy might be influenced by the enthusiasm of entities to provide communication and computational resources for forwarding announcements in which they may not be interested themselves.

Discovering Proximity Groups

The proximity discovery service enables its consumers to discover the announcements that are relevant at their current location. Mobile (and stationary) consumers discover proximities of interest and the associated event notifications will subsequently be delivered to subscribers that are located inside the proximity. Significantly, the proximity discovery service exploits the announcement concept to support mobility and to enable entities to discover proximities rather than other entities, thereby allowing them to establish communication relationships that feature a notion of anonymity in which entities are anonymous to each other but known by the middleware.

The Discovery Algorithm

The proximity discovery service hosted by either a stationary or a mobile device allows its entities to announce and discover relevant proximities and their event types according to the following discovery algorithm:

- Initially, a proximity discovery service recognises the proximities that its producers have defined.
- A proximity discovery service maintains a list describing all proximities inside which it is currently located. A specific proximity description is discarded when the hosting mobile device leaves the geographical area associated with this proximity description.
- A proximity discovery service periodically broadcasts messages announcing the proximities defined by its producers within a certain discovery area relative to its current location and the respective proximity scope. The broadcast period and the discovery area are application specific and hence, can be configured for each individual proximity discovery service.
- A proximity discovery service receiving a proximity announcement adds a proximity description to its own list if it is currently located inside the associated geographical area.

4.4.3 Mapping to Proximity Groups

The geographical scopes that can be specified by an application need to be mapped onto the underlying proximity groups. Producers map the proximities they announce to specific proximity groups, which they subsequently join in order to publish their event notifications.

Consumers discover proximities and then map proximities that have been associated with event types to which they have subscribed to proximity groups. Consequently, consumers join a proximity group of interest once they enter the associated geographical scope and leave the proximity group upon departure from the scope.

Addressing Scheme

There are two essential issues that need to be addressed when mapping announcements and subscriptions to proximity groups. Firstly, an addressing scheme for uniquely identifying groups is required and secondly, a means for consuming and producing entities to obtain the correct group identifiers needs to be provided. An approach to addressing these issues, based on statically generating a fixed number of unique and well-known group identifiers, has been described by Orvalho et al. [32]. Another approach might involve using a centralised lookup service for generating and retrieving group identifiers. However, neither of these approaches suffices for applications that accommodate a dynamically changing number of communication groups and depend on an inherently distributed architecture.

The STEAM event model exploits a decentralised addressing scheme in which identifiers representing groups can be computed from event type and proximity pairs. Each combination of event type subject and proximity (shape, dimensions, reference point, and naval location) is considered to be unique throughout a system assuming that there is no justification for applications to define multiple identical subject and proximity pairs for handling different events. A textual description of such a pair is used as stimulus for a hashing algorithm to dynamically generate hash keys that represent identifiers using node local rather than global knowledge. Upon discovery of a proximity and the associated event type, producing and consuming entities compute the corresponding group identifier if the subject is of interest. This scheme allows entities to subsequently use these identifiers to join groups in which relevant event notifications are disseminated. Moreover, it prevents entities that are not interested in certain event notifications from joining irrelevant groups and consequently, from receiving unwanted event notifications even though they might reside inside the proximity associated with a group. We have validated this concept of using node local knowledge to dynamically generate multicast group identifiers as part of our work on the mSECO extension [14] to the ECO event model.

Proximity Group Semantics

The STEAM event model exploits the message delivery semantics associated with proximity groups in order to provide end-to-end guarantees when delivering event notifications.

Proximity group semantics therefore not only serve as the basis for the semantics of the event service but also influences the set of non-functional attributes that might be supported. In order to provide strong guarantees in terms of event notification delivery reliability and timeliness, STEAM has been designed to use proximity groups that are based on a light-weight, location-aware, atomic multicast protocol for Time-Bounded Medium Access Control (TBMAC) [95]. The TBMAC protocol is based on time-division multiple access with dynamic but predictable slot allocation and has been designed for use in multi-hop ad hoc networks. It provides, with high probability, time-bounded access to the wireless medium for applications with guaranteed response time requirements. Nevertheless, STEAM might use a version of proximity groups that provides a weaker semantics and hence, imposes smaller computational overhead when accommodating applications that do not require strong end-to-end guarantees. A proximity group version providing a best-effort semantics might be based on IP multicast.

4.4.4 Mapping to Ad Hoc Networks

STEAM allows entities to define geographical scopes independently of the physical transmission range of their wireless radio transmitters. This implies that STEAM must support multi-hop event dissemination for scenarios in which proximity exceeds the radio transmission range of the sender.

Single-Hop Event Dissemination

Figure 4.16 outlines a single-hop event propagation scenario where the radio transmission range of the sender covers the entire scope of the proximity.

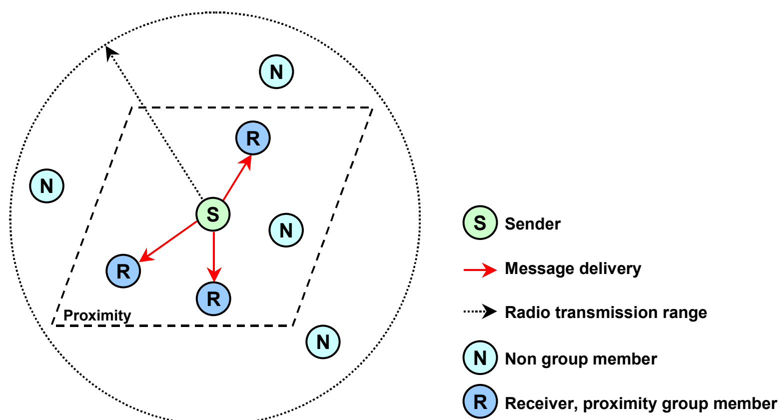


Figure 4.16. Single-hop event dissemination.

Event messages are propagated within this radio transmission range and travel exactly one hop in order to reach all potentially interested nodes. Proximity group members recognise the group identifier of the event messages that are relevant to them and may subsequently deliver these events. Nodes that are not members of such a proximity group ignore these event messages.

Multi-Hop Event Dissemination

Figure 4.17 shows a multi-hop event propagation scenario in which the proximity exceeds the radio transmission range of the sender. Proximity group member nodes must forward event messages for them to reach other members of the group. Similar to single-hop event dissemination, group members recognise the group identifier of relevant event messages and may subsequently deliver these events. However, group members also forward relevant event messages, thereby expanding their dissemination range. The maximum number of hops such event messages may travel to reach any member of the group is bounded by the proximity. Non group member nodes ignore these event messages and consequently do not forward them. Multi-hop event dissemination generally increases the range within which event messages can be propagated but characteristically imposes additional transmission latency compared to single-hop transmissions.

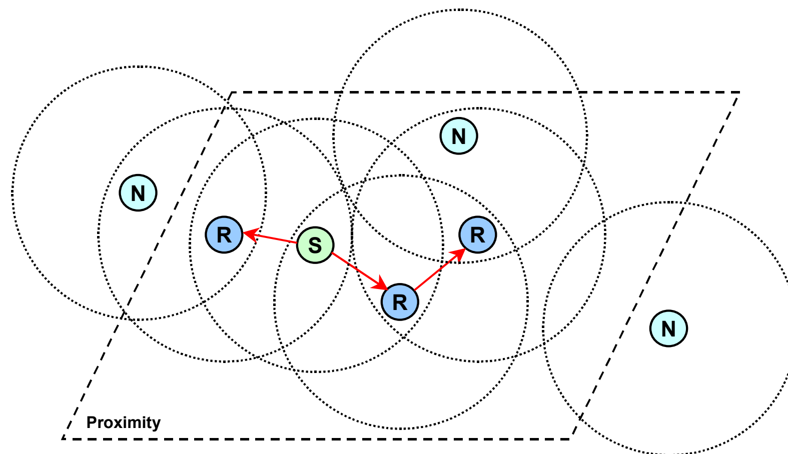


Figure 4.17. Multi-hop event dissemination.

Coverage

Coverage can be defined as the geographical area to which a particular sender can send messages using either single-hop or multi-hop communication. A proximity is said to be

covered if the associated set of member nodes is a subset of the nodes to which event messages can be propagated. Network partitions may occur in cases where an area defined by a proximity is not covered by a specific sender. They may therefore occur in the kind of multi-hop scenario illustrated in Figure 4.18, but not in single-hop scenarios where the radio transmission range covers the whole proximity.

Techniques for preventing message loss due to coverage limitations must firstly anticipate a network partition. Such partition anticipation can be based on a means for detecting link failure between individual components in ad hoc networks [96-98] or on an approach that assesses the quality of wireless connections to predict their future level of connectivity [99]. Consumers that are able to anticipate network partitions can then employ a means to recover missed event notifications once they re-establish their connections to the members of a group. Such consumers recover missed events by requesting a retransmission of previously sent events. A producer may forward the event notifications it has cached according to its persistence level to these consumers, thereby retransmitting the events that describe its current state. Producers must retransmit these events without compromising the consistency of concurrently raised event notifications.

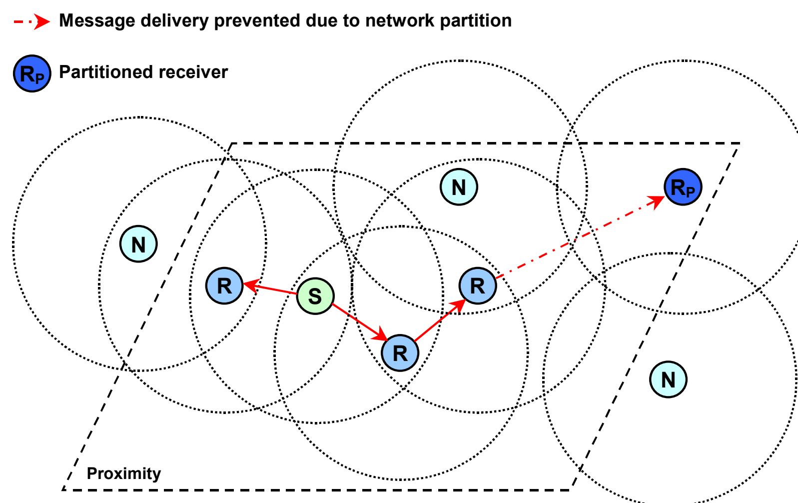


Figure 4.18. Network partitions in multi-hop event dissemination.

4.4.5 Routing Event Notifications

The concept of bounding the propagation scope of event notifications can also be exploited when routing event messages from producers to consumers in order to optimise the

propagation of specific events. STEAM uses a producer's geographical location and radio transmission range in conjunction with the proximity in which specific event notifications are to be published to transparently determine whether to use single-hop or multi-hop messages. STEAM therefore supports both single-hop and multi-hop event dissemination and typically uses cost efficient single-hop messages for publishing event notifications in proximities that are likely to be covered by a producer's wireless transmission range and employs multi-hop messages only when transmitting event notifications beyond this range.

4.5 Interface Functions

The STEAM event model provides interface functions that differ from the interfaces of other event models in that they reflect STEAM's proximity-based programming model and its distributed approach to event notification filtering.

4.5.1 Delivering Event Notifications

STEAM requires consumer applications to implement event delivery handlers and subsequently to register these handlers when subscribing. This approach enables consumers to potentially register a specific delivery handler with each of their subscriptions. Depending on their requirements, applications may therefore provide either a delivery handler for an individual subscription or for a set of subscriptions, for example for the subscriptions of a particular consumer or for a certain event type.

```
deliveryHandler {
   deliver(eventNotification n) {
   //process event notifications
   …
   }
}
```

Figure 4.19. Delivering event notifications in STEAM.

Figure 4.19 shows the STEAM interface for delivering event notifications to consumer applications. These event notifications comprise a subject that describes their respective event type. This subject can therefore be used by applications using a single delivery handler

for receiving events of different types to distinguish these types and to process individual event notifications accordingly.

4.5.2 STEAM Application Programming Interface

Figure 4.20 outlines the application programming interface supported by STEAM. These interface functions reflect the fact that STEAM is based on an implicit event model as they refer neither to explicit entities nor to designated components of any kind. Instead, the functions for announcing and subscribing to event notifications refer to an event type, the former indicating the actual type and the latter using a subject filter to name the type.

```
announce(eventType et, proximityFilter pf)
unannounce(eventType et)
subscribe(subjectFilter sf, deliveryHandler dh, contentFilter cf)
unsubscribe(subjectFilter sf, deliveryHandler dh, contentFilter cf)
raise(eventNotification n)
```

Figure 4.20. The application programming interface of STEAM.

The STEAM application programming interface also illustrates how producers and consumers specify their respective event notification filters. Producers specify their proximity filters and announce them together with their event types thereby grouping them into associated pairs while consumers specify both their subject filters and their content filters together with their delivery handlers. It is important to allow consumers to explicitly define their content filters because some applications might wish to omit this optional filter. Consumers omitting content filters express their interest in event notifications solely using their type, thereby employing a classic, topic-based subscription mechanism.

4.6 Discussion

After describing the STEAM event model for collaborative applications in detail, there are some issues remaining that we believe are essential to the rationale for the concepts presented in this thesis. We therefore discuss our approach to improve the scalability of a system together with issues related to supporting mobility in ad hoc environments.

4.6.1 Mobility

The STEAM event model addresses the challenge of accommodating mobile application components that use ad hoc networks in a number of ways. STEAM is based on an inherently distributed architecture in which the middleware is exclusively collocated with the application components and does not depend on centralised or intermediate components. Geographical scoping represents a natural way for mobile and stationary entities to identify event notifications of interest. In other words, STEAM supports mobile computing by way of its architecture and through the use of various decentralised techniques. One of the techniques that is essential to supporting mobility in STEAM is the mechanism for discovering peers, or more specifically for discovering proximities, that allows the correct producing and consuming entities to establish communication relationships. The overhead caused by this discovery mechanism is proportionally related to the frequency at which its periodic beacons are transmitted and may be significant, especially when discovering proximities in large geographical areas. The actual cost of discovering proximities is therefore being examined as part of the evaluation of STEAM presented in this thesis. This discovery frequency also influences the potential latency that entities might encounter when discovering proximities. Entities, especially those that travel at high speeds, might wish to receive beacons frequently. Hence, applications need to consider their requirements as well as the trade off between discovery overhead and potential discovery latency when configuring the discovery mechanism.

4.6.2 Scalability

A system exploiting event-based middleware for collaborative applications must be able to easily cope with a large, dynamically changing population of mobile components and the resulting dynamic reconfiguration of the connections between the components. Many of the decentralised techniques we have discovered for supporting mobility naturally accommodate changing populations and as a result, help to improve the scalability of a system. STEAM's inherently distributed architecture avoids designated middleware components that may become communication bottlenecks with increasing system scale. The concept of proximity-based event notification dissemination bounds the geographical scope within which certain information is valid and thus, limits forwarding of event notification and configuration information which may lead to a reduction in the required communication and computational resources. Combining multiple event notification filters improves the filtering precision and consequently, reduces the number of potentially unwanted event notifications being

propagated. Moreover, decentralised filtering helps to improve the scalability of a system by distributing the computational load of filter matching as a small number of filters are typically evaluated on each specific machine.

CHAPTER 5: STEAM ARCHITECTURE AND ALGORITHMS

This chapter presents a prototypical implementation of the STEAM event model, which is based on the concepts and algorithms described earlier in this thesis. We first outline the architecture of this prototype and discuss the functionality of its main components as well as the means by which these components provide for proximity-based event notifications. We then describe the key enabling algorithms and protocols that have been implemented to accommodate collaborative applications along with a demonstration program that illustrates how such applications may use STEAM.

5.1 STEAM Architecture

The STEAM architecture incorporates the use of group communication and location-awareness in order to provide proximity-based event notification dissemination that can be exploited to support mobility in collaborative applications. Such collaborative applications characteristically comprise components that are hosted by mobile devices and interact through IEEE 802.11b-based ad hoc wireless local area networks [9, 100]. Depending on the application areas in which they are used, such portable computing devices may range from handheld devices, such as personal digital assistants, to notebook computers. Some of these application areas might therefore impose requirements related to (battery) power supply and memory footprint on the middleware they use. However, addressing such requirements directly is considered to be beyond the scope of this prototype.

5.1.1 Overview

As illustrated in Figure 5.1, the architecture of the STEAM middleware essentially consists of four key components that reflect the main features of the event service. The Event Service Nucleus (ESN) implements STEAM's application programming interface and is therefore explicitly exposed to applications. The event service nucleus can be regarded as STEAM's

central component since it interconnects the remaining components and because it provides a filter engine that applies and maintains the various event notification filters that may be defined by the producing and consuming entities that comprise an application. The event service nucleus exploits a Proximity-based Group Communication Service (PGCS) for disseminating event notifications in various proximity-based multicast groups. This service is therefore responsible for routing event notifications from producers to consumers and for enforcing their delivery semantics.

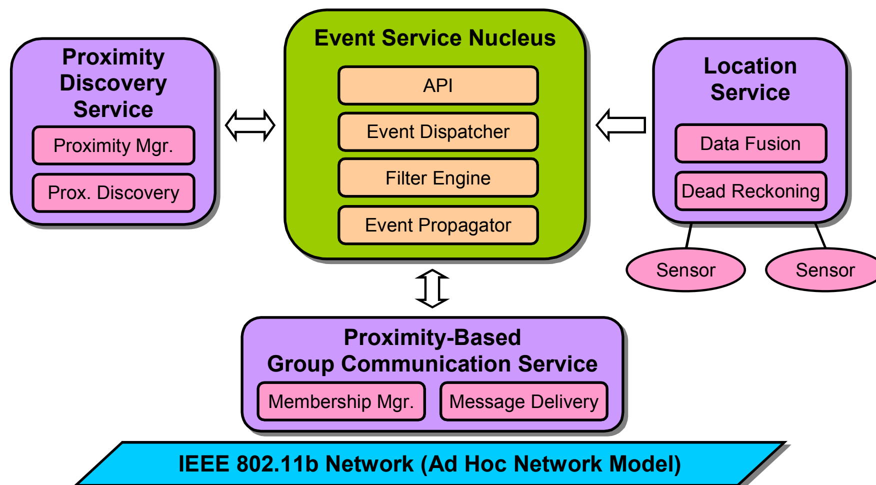


Figure 5.1. The architecture of the STEAM middleware.

The Proximity Discovery Service (PDS) provides the means for potentially mobile entities to announce and discover event notifications of interest. The discovery mechanism uses location information to map proximity scopes to the subscriptions of its consumers and as a result, manages the proximity groups that are relevant at its current location. This implies that the proximity discovery service is responsible for maintaining a consistent notion of the relationship between proximity groups and subscriptions at any given time whilst considering the migration of entities and indeed of proximities. STEAM depends on a Location Service (LS) to supply geographical location information. The location service uses sensor data to compute the current geographical location of the mobile device and subsequently provides this location information to the middleware and to event producers and consumers hosted by the device.

5.1.2 Distribution

Figure 5.2 illustrates that every mobile device has identical STEAM capabilities. These capabilities are implemented by the event service nucleus and by the services providing proximity-based group communication, proximity discovery, and location information that run on each machine. Applications may connect a variable number of consumers and producers to the middleware on each mobile device, thereby allowing individual devices to initiate and participate in one or more event-based communication sessions. Each of these entities regulates the propagation of its proximity-based event notifications by exploiting the concepts of announcements and subscriptions. STEAM augments these well-known and widely-used concepts in order to support scoped event notification dissemination and ultimately mobility. The event type and proximity information announced by producers is exploited by the discovery mechanism primarily to establish communication relationships between mobile entities rather than to optimise event notification routing. Subscriptions are used *locally* to map consumer interests to the sets of currently available event notifications being disseminated within the discovered proximities.

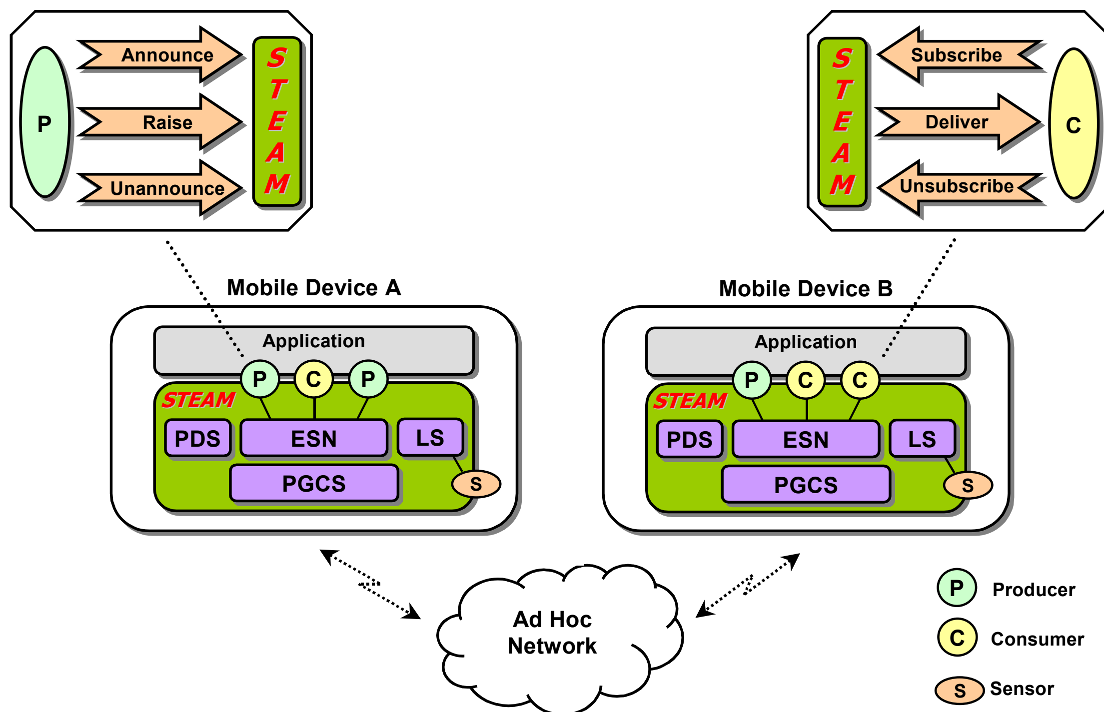


Figure 5.2. Mobile devices hosting STEAM middleware.

5.1.3 Proximity-Based Event Notification Management

STEAM's approach of exploiting proximity to support mobility in collaborative applications affects all the mechanisms used to handle event-based communication. The various decentralised techniques used by STEAM are incorporated in its means of announcing, subscribing to, raising, and delivering event notifications, as described in the following sections.

Announcing and Discovering Proximity-Based Event Notifications

The means of announcing and discovering proximity-based event notifications implemented by the STEAM middleware allows every mobile device to maintain an event type and proximity filter repository on behalf of its entities. This repository is part of the event service nucleus and contains the filters that describe the proximities that are available at the device's current location as well as type information describing the events that are associated which these proximities.

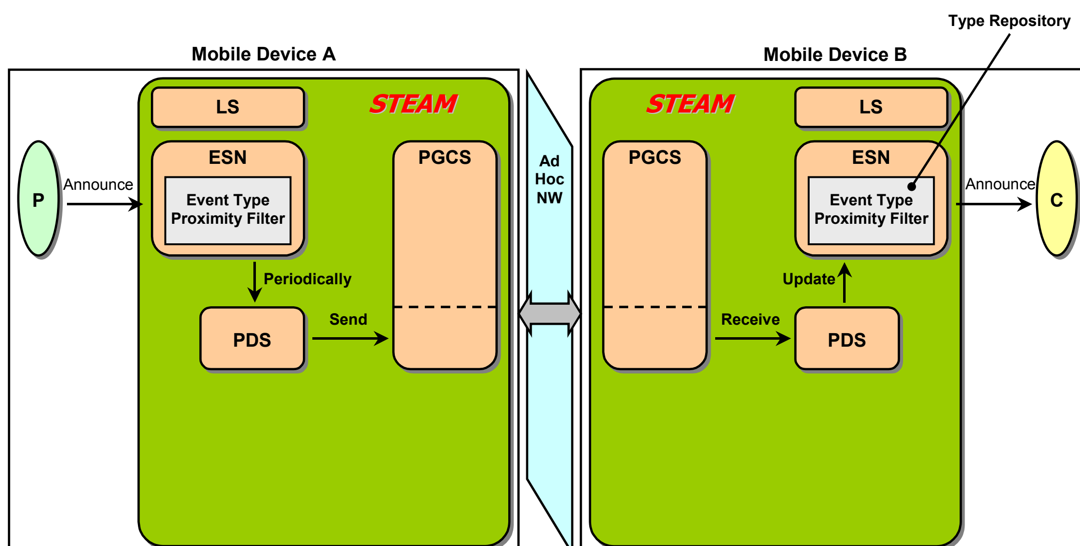


Figure 5.3. Announcing and discovering proximity-based event notifications.

Figure 5.3 illustrates this discovery mechanism by outlining an example in which a mobile device (A) disseminates announced information on behalf of a producer to another mobile device (B) that hosts a consumer. Initially, the newly announced event type and proximity filter pair is added to the local event type repository of device A. The proximity discovery service of this device subsequently disseminates this information periodically in its vicinity

using a well-known “discovery” proximity group. The discovery service of device B, which is located nearby device A, eventually receives these beacons and updates its own repository with this information. Newly discovered event type information is also made available to the application using an event announcement delivery handler. In general, the proximity discovery mechanism announces and discovers event types and proximities on behalf of its entities, regardless of them being producers or consumers. Every discovery service periodically assesses its event type repository, thereby disseminating the type-proximity pairs that have been announced by its producers and discarding the proximities that have expired, i.e., that no longer cover its current location.

Subscribing to Proximity-Based Event Notifications

Subscribing consumers cause the middleware to map subscription information to the set of currently available proximities and as a result, may cause the event service to join one or more proximity groups and subsequently to deliver the event notifications disseminated in these groups. Figure 5.4 illustrates this subscription mechanism and shows that subscriptions issued by consumers on a mobile device are maintained in another local repository. These subscription repositories therefore contain the subject filter, content filter, and delivery handler triples supplied by their subscribers. The subject filter of a newly issued subscription is applied to the names of the event types maintained in the event type repository. A matching subject filter identifies an event type and proximity pair of interest and results in the middleware joining the proximity group that corresponds to this pair. Notably, subscriptions are maintained locally and hence, do not require transmission of subscription messages.

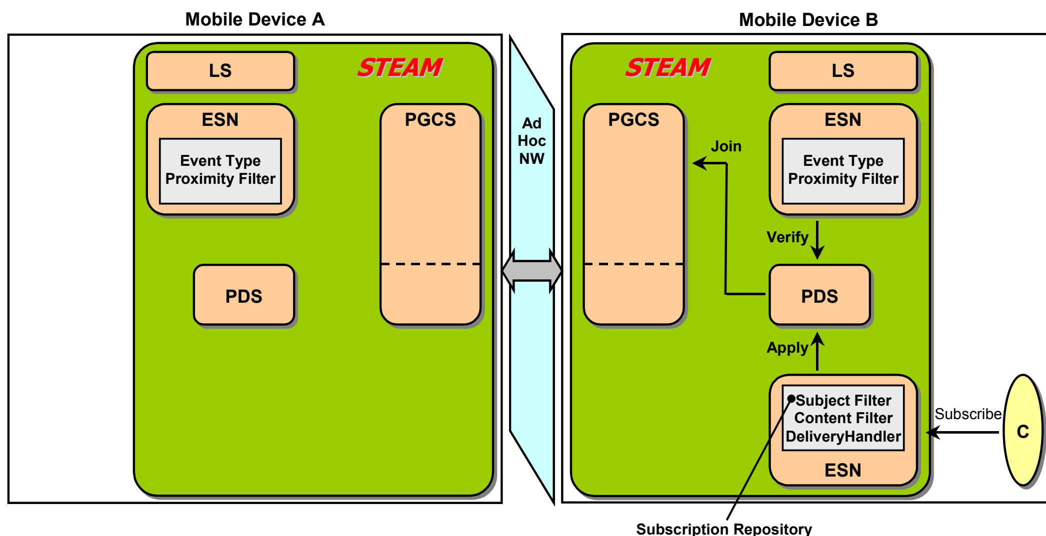


Figure 5.4. Subscribing to proximity-based event notifications.

A proximity discovery service verifies the mapping of subscriptions to event types upon updating its event type repository. This may lead to a change in the set of proximities of interest and consequently may cause a change in proximity group memberships. Such membership changes may apply when a mobile device enters or leaves the scope of a proximity, which may lead to the addition of a new membership or to the cancellation of an existing one.

Raising Proximity-Based Event Notifications

Figure 5.5 outlines an example in which a producer hosted by mobile device A raises event notifications. The subject of these event notifications is used to retrieve the event type and proximity pairs that describe the scopes in which a specific event notification is to be disseminated. Once these pairs have been identified, the associated group identifiers can be determined and the event notification can be propagated using the correct proximity groups. In general, we expect an individual event notification to be disseminated in a single proximity group. However, some applications might require specific event notifications to be propagated simultaneously in multiple, overlapping proximity groups. Consumers that have subscribed to such proximity groups may consequently receive multiple copies of these event notifications as each membership may result in the delivery of their event notifications.

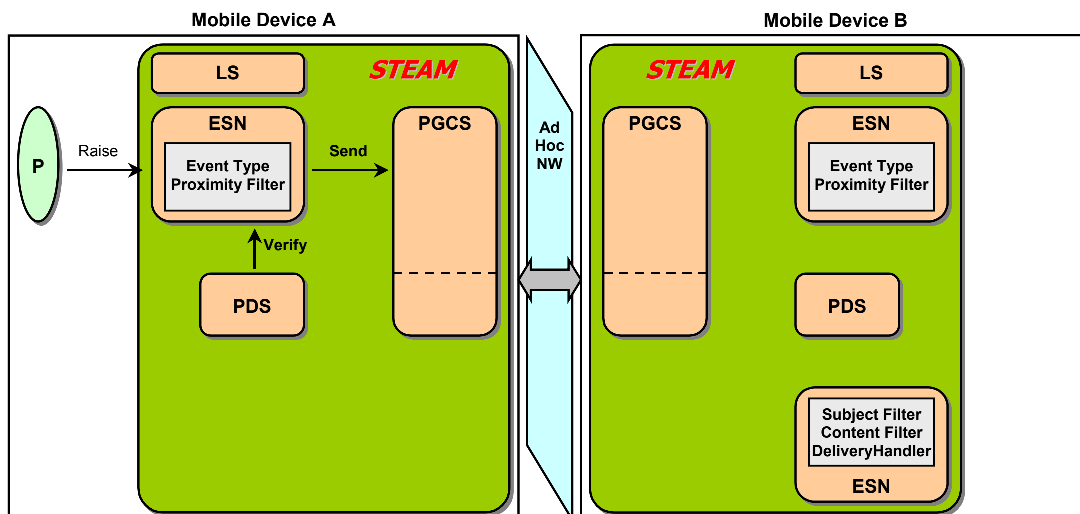


Figure 5.5. Raising proximity-based event notifications.

Delivering Proximity-Based Event Notifications

The STEAM event service receives event notifications propagated in proximity groups it has previously joined. Mobile device B, shown in Figure 5.6, illustrates the means of delivering such event notifications. Event notifications received in a particular proximity group are verified to conform to the corresponding event type before their content is matched. In other words, the subject of an event notification can be used to retrieve its type information from the type repository and to obtain the correct content filter(s) and delivery handler(s) from the subscription repository. These delivery handlers are then used to pass event notifications that match their content filters to the application.

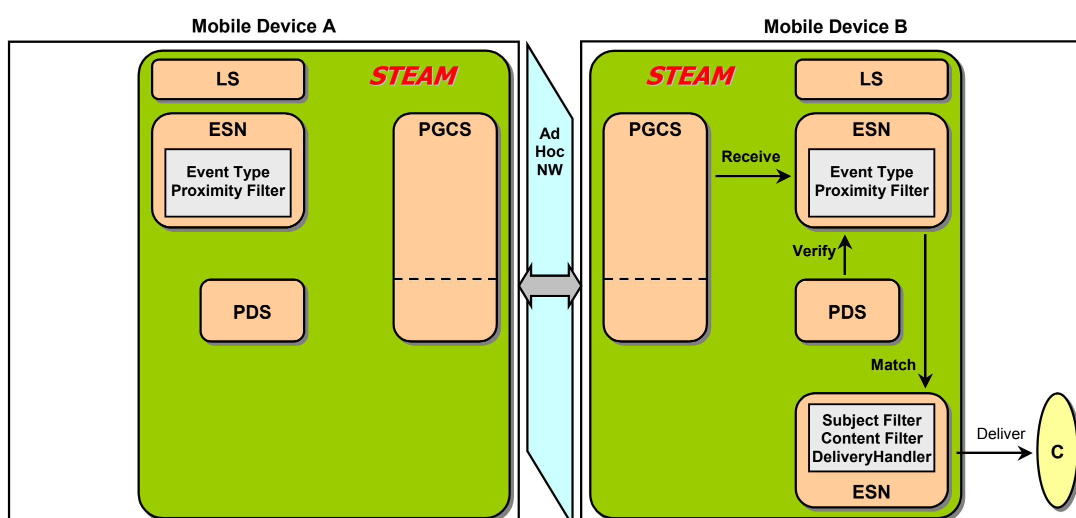


Figure 5.6. Delivering proximity-based event notifications.

5.2 The STEAM Event Service

Applications connect to the STEAM event service by obtaining device-local entity objects that provide either a producer or a consumer interface. This allows applications to connect a variable number of entities to the middleware on each mobile device.

5.2.1 Event Notifications and Event Types

STEAM's event notifications are typed since they are described by an expressive structure into which a set of functional and non-functional attributes can be mapped. An individual

event notification is therefore defined firstly by a type describing its structure and secondly by an instance describing the specific values of its attributes. This approach of explicitly defining the type of event notifications enables the middleware to exploit type information in order to verify that event notifications conform to their respective types.

Event Types

Producers define event types as objects that comprise subject, content, and a non-functional attribute list. The event type content can be specified as a set of a variable number of parameters in which each parameter is of a particular type. The set of currently available types contains commonly used types, such as string, integer, and floating point, as well as location and time types that describe the context of event notifications. The location type in particular is essential for supporting collaborative applications that exploit proximity-based event notifications.

```
S_EventParameterDeclaration* epd;
SP_dsEventType* et;
const int numParameter = 5;

epd = new S_EventParameterDeclaration[numParameter];
epd[0] = S_EventParameterDeclaration("String", S_STR);
epd[1] = S_EventParameterDeclaration("Integer Number", S_INT);
epd[2] = S_EventParameterDeclaration("Floating Point Value", S_DBL);
epd[3] = S_EventParameterDeclaration("Geographical Location", S_POS);
epd[4] = S_EventParameterDeclaration("Time", S_TIM);
et = new SP_dsEventType("Example Subject", numParameter,
                         &epd, SP_SINGLE_HOP);
```

Figure 5.7. Defining the structure of an event type.

The example shown in Figure 5.7 illustrates how STEAM allows producers to define the structure of an event type as an object. This example shows an event type comprising subject and content, the former defining a character string value and the latter defining a set of five parameter name-type pairs that reflect the currently available parameter types. Applications may therefore define event types by dynamically defining content structures that address their requirements.

Event Notifications

Once an event type has been defined, producers may instantiate event notifications that conform to the structure of this event type. Figure 5.8 shows an example of an event notification of the previously defined type. Note that the application explicitly provides the

values for the parameters describing context. A producer application may retrieve the current time from the underlying operating system and the actual location information by means of the location service provided by STEAM. Once such an event notification has been propagated and delivered, a consumer application may access its content using the operators of the event notification object.

```
SP_dsEvent* ei;
S_ParameterValue* pv = new S_ParameterValue[numParameter];

pv[0] = SP_ParameterValueSTR("Some text");
pv[1] = SP_ParameterValueINT(1500);
pv[2] = SP_ParameterValueDBL(123.456);
pv[3] = SP_ParameterValuePOS(5320.0N, 615.0W);
pv[4] = SP_ParameterValueTIM(Mon Aug 25 12:25:46 2003);
ei = new SP_dsEvent(et->subject(), numParameter, &pv, *et);
```

Figure 5.8. Instantiating an event notification.

Supported Attributes

The current version of the STEAM event service implementation supports a range of functional and non-functional event notification attributes. These attributes include subject and content of event notifications as well as location and time attributes that may be used to describe the context of certain event notifications. Applications may explicitly associate context attributes with event instances by defining them as part of the content structure. In addition, the middleware will implicitly associate a producer and a forwarder location attribute with every event notification that is being disseminated in order to enable geographical filtering. As these location attributes are transparently managed by the middleware, applications are neither required to define nor to instantiate them. Furthermore, STEAM features an attribute that allows applications to select the routing strategy that best suits their requirements. A shown in Figure 5.7, such attributes can be associated with individual event types and specify whether related event notifications are propagated using single-hop or multi-hop messages. Support for other attributes, including attributes describing event notification delivery semantics and quality of service requirements has not been implemented yet. As a result, all event types are implicitly classified as non real-time.

Type Safety

This approach of explicitly defining the type of event notifications also enables the middleware to provide some notion of type safety for its event notifications without relying on

a pre-processor. Type information can be used to verify that the content of event notifications conform to their respective types. Such type checks may be performed at run time, for example when event notifications are instantiated by producers. Furthermore, type information is essential for unmarshalling event notifications that have been serialised for transmission. The middleware can use such type information to create consumer side objects describing event notifications that have been disseminated and to verify that these event notifications are of the correct structure.

5.2.2 Event Notification Filters

STEAM supports the three essential event notification filters. Subject filters contain a string value that is matched against the name of an event type. Content filters comprise a set of expressive and self-describing filter terms that can be applied to the content parameters of event notifications. Proximity filters describe a two dimensional geographical area that specifies either the stationary or mobile scope within which event notifications are propagated.

Content Filters

Consumers may define content filters that contain a variable set of either conjunctive or disjunctive filter terms. One or more of these terms may be applied to each event notification parameter. For example, two terms containing magnitude operators might be applied to a numeric parameter in order to filter a parameter range. This dynamic content filter structure combined with the set of available operators enables applications to define their filter requirements in an expressive manner compared to approaches where the number of filter terms depends on the number of event notification parameters, for example as proposed by JEDI.

S_INT, S_DBL, S_TIM operators : "==", "<>", "<", ">", "<=", ">="

S_STR operators : "==", "<>"

S_POS operators : DISTANCE_INCREASES, DISTANCE_DECREASES,
WITHIN_RANGE, BEYOND_RANGE

Figure 5.9. Content filter term operators.

As summarised in Figure 5.9, the set of operators that can be applied to a particular parameter depends on its type. STEAM supports operators for parameter types describing context. This allows applications to exploit the context of the producer sending an event notification together with the consumer context when determining content filter constraints. For example, location-aware consumers can use their actual location when applying range operators on location parameters instantiated by producers. Furthermore, applications may exploit historical context information when filtering event notifications. Consumers may define distance operators that use their previous and current location in order to determine whether they are moving towards or indeed away from a certain producer. Note that the matching result of such filter terms might be inconclusive, for example when such terms are applied to stationary consumers. Such inconclusive filter terms are excluded when computing the matching result of a content filter.

Proximity Filters

The example shown in Figure 5.10 illustrates how filters referring to stationary and mobile proximities, which will eventually be mapped onto absolute and relative proximity groups respectively, may define either circular or rectangular areas. These shapes are defined by their dimensions and by a reference point located inside their area. For example, circular shapes are defined by a radius and by a reference point that is naturally located at the centre of the circle. Proximity filters then map these reference points to either a fixed or a moving naval, depending on whether they are stationary or mobile. For example, the rectangular proximity filter maps its reference point to a fixed naval whereas the circular filter defines a mobile proximity which is implicitly mapped to a naval represented by the actual location of its producer. Regardless of their shape, proximity filters provide an `inside` function that enables the middleware to verify whether an entity resides within the specified area.

```
SP_Shape* rectangle;
SP_Shape* circle;
SP_ProximityFilter* pf_r;
SP_ProximityFilter* pf_c;
SLS_Location* naval = new SLS_Location(5320.0N, 615.0W);

rectangle = new SP_Rectangle(x_dim, y_dim, x_ref, y_ref);
pf_r = new SP_ProximityFilter(rectangle, SP_ABSOLUTE, naval);

circle = new SP_Circle(radius);
pf_c = new SP_ProximityFilter(circle, SP_RELATIVE);
```

Figure 5.10. Instantiating stationary and mobile proximity filters.

5.2.3 Repositories

Decentralised repositories maintain announcement and subscription information on behalf of the entities located on specific mobile devices. Type repositories handle event types and proximity filters on behalf of both producing and consuming entities while subscription repositories manage subject filters, content filters, and event notification delivery handlers in support of their consumers. Repositories hosted by a certain mobile device maintain only information that is relevant to this device. Type repositories exclusively manage event types and proximity filters that are relevant at the location of a device and are of interest to the device's entities, while subscription repositories only handle subscriptions of local consumers.

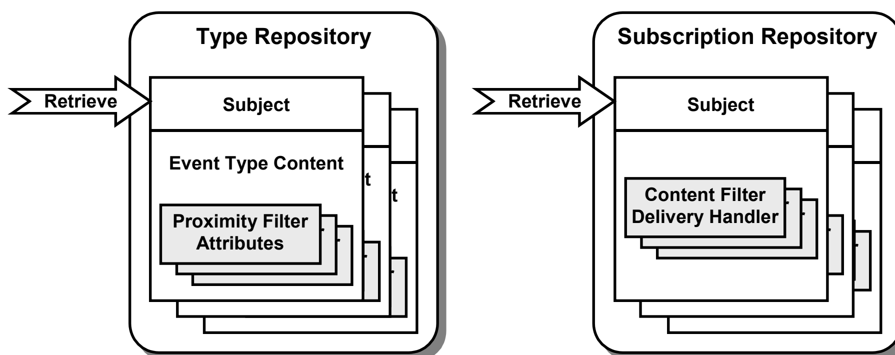


Figure 5.11. Maintaining announcement and subscription information.

Figure 5.11 illustrates the organisation of these repositories, which enables the middleware to retrieve information related to a certain event type. Type repositories hold the event type content as well as a list of the proximity filters and attributes that are associated with a specific subject whereas subscription repositories contain the lists of content filters and delivery handlers that relate to some subject. This organisation scales well as it can easily cope with a variable number of event types and event filters and is flexible in terms of catering for potential extensions required by future versions of STEAM. For example, type repositories might need to be extended to maintain attributes that are specific to a subject rather than to a proximity filter. Both repositories have been implemented as hash tables in order to allow for an access mechanism with a constant retrieval time.

5.2.4 Addressing Scheme

STEAM's distributed addressing scheme replaces the kind of centralised approach traditionally used for identifying peers of interest and therefore represents a key enabling

mechanism to STEAM's inherently distributed architecture. This addressing scheme enables mobile devices to recognise proximities of interest and to *locally* compute proximity group identifiers from event type and proximity information. This technique is based on using serialised descriptions of individual event type and proximity pairs, which are considered to be unique throughout a system, as the stimuli for a hashing algorithm generating group identifiers.

Collisions

Depending on a number of factors, including the quality of the hash function and the ratio of stimuli to potential identifiers, this approach might lead to colliding identifiers. Such collisions occur when two distinct stimuli x and y generate the same identifier. In other words, there exist stimulus pairs, such that $x \neq y$, for which $h(x) = h(y)$. Collisions may result in different kinds of event notifications being disseminated in proximity groups using the same identifier. This does not affect a system provided that such proximity groups occupy different geographical scopes, i.e., that groups do not overlap. Overlapping proximity groups with colliding identifiers can lead to unwanted event notifications being received by certain mobile devices. However, such devices will be prevented from delivering unwanted event notifications to their applications as the middleware's run-time type checking mechanism detects and eventually discards these event notifications. Hence, colliding group identifiers may lead to additional use of communication and computational resources, but will not cause delivery of unwanted event notifications.

Computing Group Identifiers

The STEAM middleware uses a hash algorithm that generates 24 bit group identifiers from variable length character strings. The implemented algorithm is based on a combination of a hash function for such stimuli proposed by Preiss [101] with the approach of using a hash function multiplier [102] in order to reduce the chance of collisions. Figure 5.12 depicts the structure of the character strings that describe event type and proximity pairs. Such strings comprise the subject of event types and the particulars of proximity filters. Proximity filters are described by their shapes, by an indicator for stationary and mobile scopes, and by the coordinates that specify the location of the filters' navals. Note that a mobile proximity filter always describes its initial naval location since its actual naval location might change over time therefore enabling mobile devices to generate consistent group identifiers.

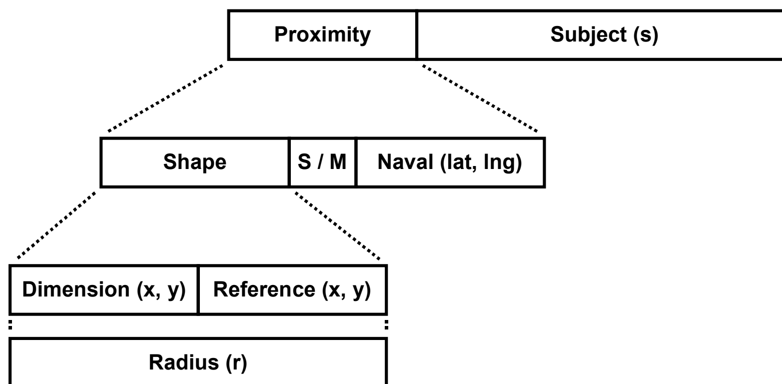


Figure 5.12. Computing group identifiers from event type and proximity pairs.

Application Example

The application illustrated in Figure 5.13 and Figure 5.14 shows a producer and a consumer implementing a traffic light scenario in which a light disseminates its status to approaching vehicles. Figure 5.13 outlines the producer application generating event notifications on behalf of a traffic light. After starting STEAM and a location service, the producer defines a “Traffic Light” event type that describes the light’s status and location as well as a stationary proximity. This proximity surrounds the traffic light and uses single-hop messages when disseminating event notifications. The producer then announces its event type and proximity filter pair before instantiating and raising event notifications that conform to this event type.

```
//start location service and STEAM
SLS_SteamLocationService* sls = new SLS_SteamLocationService();
S_Steam* steam = new S_Steam(*sls);

//obtain producer entity
SP_SteamProducerEntity* sp = steam->obtainProducerEntity();

//define event type and proximity filter
S_EventParameterDeclaration* epd;
SP_dsEventType* et;
SP_Shape* cir;
SLS_Location *cLoc;
SP_ProximityFilter* pf;
const int nParameter = 2;

epd = new S_EventParameterDeclaration[nParameter];
epd[0] = S_EventParameterDeclaration("Status", S_STR);
epd[1] = S_EventParameterDeclaration("Location", S_POS);
et = new SP_dsEventType(“Traffic Light”, nParameter, &epd, SP_SINGLE_HOP);
cir = new SP_Circle(50.0);
sls->position(cLoc); //retrieve current location
pf = new SP_ProximityFilter(cir, SP_ABSOLUTE, *cLoc);
```

```
//announce event type and proximity filter
sp->announce(*et, *pf);

//instantiate event notification
SP_dsEvent* ei;
S_ParameterValue* pv = new S_ParameterValue[nParameter];

pv[0] = SP_ParameterValueSTR("Green");
pv[1] = SP_ParameterValuePOS(*cLoc);
ei = new SP_dsEvent(et->subject(), nParameter, &pv, *et);

//raise the event notification
sp->raise(*ei);
```

Figure 5.13. A producer generating event notifications.

Figure 5.14 shows the consumer application acting on behalf of a vehicle. The consumer implements its event notification delivery handler and starts STEAM and a location service in a manner similar to the producer. The consumer then defines a content filter that uses geographical location information to filter out "Traffic Light" event notifications unless the vehicle is driving towards the traffic light.

```
//implement delivery handler
class Vehicle_cbdImplementation : public SC_CallbackDelivery {
public:
  void deliver(SC_dsEvent*& ei) {
    //process event notification
    char* status = ei->parValSTR(0);
    ..
  }
}

//start location service and STEAM
SLS_SteamLocationService* sls = new SLS_SteamLocationService();
S_Steam* steam = new S_Steam(*sls);

//instantiate delivery handler implementation
Vehicle_cbdImplementation* dh = new Vehicle_cbdImplementation();

//obtain consumer entity
SC_SteamConsumerEntity* sc = steam->obtainConsumerEntity();

//define content filter
SC_ConjunctiveContentFilter* cf = new SC_ConjunctiveContentFilter();
cf->addTermPOS(1, SC_POS_DISTANCE_DECREASES);

//subscribe consumer
sc->subscribe("Traffic Light", *dh, cf);

//receive event notifications
..
```

Figure 5.14. A consumer receiving event notifications.

5.3 Discovering Proximities

The proximity discovery mechanism allows entities to announce and discover event type and proximity filter pairs that describe either stationary or mobile scopes. This mechanism uses the type repository for maintaining the set of event types and associated proximity filters that is of interest to the entities hosted by a certain mobile device. These event type and proximity filter pairs together with the subscriptions issued by local consumers represent the functional and geographical aspects that determine proximity group membership.

The proximity discovery mechanism essentially comprises two algorithms, one for disseminating event type and proximity filter pair announcements and another for handling such announcements upon receiving them. Both algorithms operate on the type repository of a particular mobile device and collaborate in order to maintain the relevant locally and remotely specified event type and proximity filter pairs. Hence, type repositories are maintained according to geographical relevance and are independent of the set of issued subscriptions.

5.3.1 Announcing Event Types and Proximity Filters

Figure 5.15 describes the algorithm that is used to announce and to a certain extent maintain the event type and proximity filter pairs on a particular mobile device. The algorithm periodically traces through all event type and proximity filter pairs stored in a type repository in order to announce the pairs that have been defined by local producers and to verify the geographical validity of proximity filters. Hence, the algorithm essentially maintains stationary and mobile proximity filters that are relevant at the actual location of a mobile device and periodically announces filters with a locally specified naval.

Stationary Proximity Filter

As shown in Figure 5.15, stationary proximity filters remain in a repository as long as their scope includes the current location of the mobile device regardless of whether they have been specified locally or remotely. An individual proximity filter is removed from the repository once the device has left the associated scope. Removing a proximity filter may lead to a change to the set of proximity groups that the device has previously joined. Any group membership associated with such a filter is cancelled regardless of whether functional interest, expressed by related subscriptions, remains.

Mobile Proximity Filter

The dynamic aspect of mobile proximity filters causes some variation in the means by which such filters are maintained and announced. Mobile proximity filters specified by local producers always (by definition) include the actual location of the mobile device and migrate together with the device. Consequently, the migration of a mobile device does not cause such proximity filters to expire. These filters can therefore be announced without verifying their validity. In addition to enabling remote devices to discover mobile proximity filters, these announcements provide a means for disseminating location updates describing the migration of mobile filters. Every mobile proximity filter is therefore updated with the latest device location prior to being announced. The validity of remotely specified mobile proximity filters is verified when updates on their latest locations are discovered. This prevents validity checks using cached and therefore potentially obsolete (due to migration) location information.

```
for (all event type and proximity filter pairs in the repository) {
    if (proximity filter is stationary) {
        if (current location is inside the proximity) {
            if (naval is defined locally) {
                announce event type and proximity filter pair
            }
        } else {
            remove proximity filter from repository
            update proximity group membership
        }
    }
    else if ((proximity filter is mobile) and (naval is defined locally)) {
        update naval location with current location
        announce event type and proximity filter pair
    }
}
```

Figure 5.15. Announcing event type and proximity filter pairs.

5.3.2 Discovering Event Types and Proximity Filters

Figure 5.16 describes the algorithm that handles event type and proximity filter pairs upon their reception. The algorithm adds newly discovered event types and proximity filters to the type repository of any mobile device residing inside the proximity filter's scope regardless of the set of available subscriptions. In addition, proximity group membership is updated for subscriptions that map to the event types of newly discovered proximity filters. On the other hand, known proximity filters, whose scopes no longer include the device's location, are removed from the repository and associated proximity group memberships are cancelled. Other event type and proximity filter pairs describe information that is either irrelevant at the current location or already known and are therefore discarded.

Stationary and Mobile Proximity Filters

Remotely specified event type and proximity pairs are handled similarly regardless of whether they describe stationary or mobile proximity filters. Both stationary and mobile proximity filters that are no longer relevant at the current location are removed from the repository. Mobile filters are removed based on their latest naval location while being identified according to their initial naval location. This enables the middleware to identify mobile proximity filters even though their actual naval location changes.

```
receive event type and proximity filter pair
if ((event type and proximity filter pair is not already in the repository) and
    (current location is inside the proximity))
    update repository with event type and proximity filter pair
    update proximity group membership
}
else if ((event type and proximity filter pair is already in the repository) and
    (current location is not inside the proximity))
    remove proximity filter from repository
    update proximity group membership
}
else {
    discard received event type and proximity filter pair
}
```

Figure 5.16. Discovering event type and proximity filter pairs.

5.3.3 Maintaining Event Types and Proximity Filters

Figure 5.15 and Figure 5.16 show how event types and associated stationary and mobile proximity filters are announced, discovered, and maintained. These figures imply that proximity filters may be added and eventually removed from a particular type repository whereas event types are persistently maintained once they have been discovered even in the absence of corresponding proximity filters. Table 5.1 summaries the approach reflected by the discovery mechanism for maintaining proximity filters. The geographical validity of locally defined filters as well as of remotely specified stationary filters is verified periodically whereas remotely specified mobile proximity filters can only be checked upon receiving announcements with the latest naval updates.

	Periodically Maintained (when announced)		Maintained Upon Discovery	
	Locally Specified	Remotely Specified	Locally Specified	Remotely Specified
Stationary Proximity Filter	✓	✓	×	✓
Mobile Proximity Filter	✓	×	×	✓

Table 5.1. Maintaining proximity filters.

5.3.4 Discovery Range

The proximity discovery service periodically disseminates announcements comprising event type and proximity filter pairs inside the geographical area specified by the respective proximity filter. In addition to this producer perspective on the discovery scope, applications may wish to impose a notion of discovery scope that is driven by a consumer’s point of view. For example, a mobile device in a smart environment may only wish to discover available services on behalf of its user once they are within a certain range. Consumers hosted by a specific mobile device describe their area of interest by defining a discovery range that is applied by the proximity discovery service when receiving announcements. Consequently, the scope of proximity discovery is determined by combining the areas defined by a proximity and by a consumer discovery range. Event type and proximity filter pairs are only discovered at locations included in both scopes as the discovery mechanism firstly applies its discovery scope and then, as shown in Figure 5.16, verifies the proximity area.

5.4 Disseminating Event Notifications and Announcements

Both event notifications and announcements are disseminated using the proximity-based group communication service. An individual proximity group essentially comprises a multicast group and an associated geographical scope. Entities join and leave multicast groups according to their location, i.e., they join when entering and leave upon departing the associated scope, and use the filters that describe these scopes as well as the types of the propagated information to verify geographical and structural correctness of the messages they receive.

Multicast groups disseminating event notifications are functionally and geographically described by the associated event type and proximity filter pair while groups disseminating announcements are described by the consumer discovery range and a type describing announcement messages. In fact, a well-known combination of a single multicast group with an announcement type is used when propagating announcement messages on behalf of the proximity discovery service. However, proximity-based multicast groups essentially route both event notifications and announcements from a sender to multiple receivers using either single-hop or multi-hop messages.

5.4.1 Proximity-Based Multicast Groups

The proximity-based group communication service uses the group identifiers generated from event type and proximity filter pairs as addresses for its multicast groups. These proximity-based multicast groups enforce the semantics associated with message delivery and implement the programming interface provided by the proximity-based group communication service.

Dissemination Semantics

The current version of the STEAM event service uses a proximity-based group communication service that is based on IP multicast. Exploiting IP multicast for disseminating messages provides best-effort delivery semantics, which does not guarantee that any subscriber will necessarily receive a specific event notification nor that an individual announcement will be received by nearby devices. The current implementation of STEAM therefore provides a best-effort event service but has been designed to use proximity groups that are based on the TBMAC protocol [95]. Hence, a future version of STEAM will provide strong guarantees in terms of event notification delivery reliability and timeliness.

Programming Interface

Figure 5.17 summarises the programming interface supported by the proximity-based group communication service. This interface differs from other group communication interfaces due to the fact that it supports proximity-based message dissemination. Joining as well as leaving a specific proximity group requires a particular description of a proximity, known as a proximity filter cell, that includes the actual proximity filter as well as the associated event type subject in addition to the group identifier. Messages may be disseminated in a specific group using the `send` operation while specifying the nature of these messages as either single-hop or multi-hop. The `receive` operation blocks until either a single-hop or a multi-hop message becomes available in any of the previously joined proximity groups. Because they are based on IP multicast, these proximity groups support a membership mechanism that only requires consumers to join proximity groups in order to receive messages while producers may disseminate messages without being group members. Moreover, specific proximity groups may be joined or indeed used without explicitly creating them. Hence, the STEAM middleware maintains membership only on behalf of consumers and is not concerned with creating and deleting individual proximity groups.

```
class SPGC_ProximityGroupCommunication {
public:
  //join a specific proximity-based multicast group
  void join(SPGC_GroupId& id, SPGC_ProximityFilterCell* pfc);

  //leave a specific proximity-based multicast group
  void leave(SPGC_GroupId& id, SPGC_ProximityFilterCell* pfc);

  //disseminate either a single-hop or a multi-hop message in a specific
  //proximity-based multicast group
  void send(SPGC_GroupId& id, SP_PropagationType pt, const String msg);

  //receive either a single-hop or a multi-hop message disseminated in
  //any of the previously joined proximity-based multicast groups
  void receive(String msg);

  //instantiate proximity groups and set routing optimisation mode
  SPGC_ProximityGroupCommunication(double gossip_p, double gossip_d);
};
```

Figure 5.17. The programming interface of the proximity-based group communication service.

5.4.2 Routing Messages

STEAM implements a routing protocol that exploits proximity to control multicast-based flooding of the underlying ad hoc network. This approach provides a means for disseminating

messages in a one to many manner within the boundaries defined by a proximity without introducing extra overhead for maintaining routing information. Such routing information characteristically needs to be updated more frequently with increasing speed of mobile devices and thus, approaches that attempt to maintain routing information are less suited for applications comprising highly mobile entities [103]. Other, well-known routing protocols for wireless ad hoc networks, such as AODV, DSR, TORA, and DSDV, which have been discussed and compared in [104-106], focus on peer to peer routing rather than on multicasting.

Single-Hop and Multi-Hop Routing

Messages may be disseminated using either a single-hop or a multi-hop routing protocol. Single-hop messages are propagated in their respective proximity groups within the physical radio transmission range of the sending device. Devices that are members of such groups can receive these messages and may subsequently deliver them to their applications without having to forward them. Multi-hop messages are propagated using a variation of flooding in which messages are forwarded in multicast groups and within the boundaries of their proximities. Member devices use sequence numbers to identify new messages that they need to forward. These sequence numbers are based on a combination of a device's IP address, which we assume to be available, and a sender-side message counter that makes them unique. The sequence numbers of forwarded messages are temporarily stored in a sliding window buffer thereby preventing multiple forwarding of individual messages.

Forwarding Strategy

Our approach of routing multi-hop messages in multicast groups implies that devices only forward messages in which they are interested. Multicast group members forward their messages while other devices residing in the associated scope will neither receive nor forward these messages. This applies to multicast groups disseminating event notification messages and announcement messages alike. However, since all devices need to discover proximities and use the well-known discovery group when doing so, all devices located in a certain proximity effectively forward announcement messages even if they are not interested in the actual event type and proximity described by a specific message.

Receiving Single-Hop and Multi-Hop Messages

Figure 5.18 describes the algorithm used by the `receive` operation shown in Figure 5.17 when handling singe-hop and multi-hop messages. The algorithm initially obtains certain data

from the message header, including sender location, forwarder location, message sequence number, and proximity identifier. This identifier is subsequently used to verify that the structure of the received messages conforms to the expected message type and to retrieve the correct proximity filter. The sender location is used when checking the proximity of mobile groups, thereby ensuring that the latest naval location is utilised. This additional group membership check prevents unwanted messages, for example received due to colliding group identifiers, from being delivered. The algorithm forwards multi-hop messages after updating their forwarder location with the current location and returns these newly received messages to its event service.

```
receive(msg)
//extract sender location, forwarder location, sequence number, and proximity identifier
deserialise(msg, sLoc, fLoc, seq, pKey)
//check simulated transmission range
if (simulatedTransmissionRange.inside(cLoc, fLoc)) {
    //verify subject and retrieve relevant proximity filter
    if verify(pKey) {
        pf = retrieve(pKey)
        //single-hop or multi-hop message
        multi_hop = foo(seq)
        //handle single-hop or new multi-hop message
        if ((not multi_hop) or (multi_hop and not slidingWindow.seen(seq))) {
            inside = pf.inside(cLoc, sLoc)
            if (inside and multi_hop) {
                //forward multi-hop message
                updateForwarderLocation(msg, cLoc)
                send(pKey.groupId(), msg)
                slidingWindow.insert(seq)
            }
            if (inside) return msg
        }
    }
}
```

Figure 5.18. Receiving single-hop and multi-hop messages.

Optimisations Techniques

Depending on the number and the distribution of participating devices, flooding-based multi-hop routing can produce a large number of forwarded messages and may result in individual devices receiving several copies of the same message. We have addressed this by extending the algorithm described in Figure 5.18 with a gossip-based optimisation technique [107]. Such gossip-based approaches reduce the overhead of flooding-based routing protocols by assigning some message forwarding probability to each device. Applications can specify gossip parameters Gossip(d, p), which are then applied by the algorithm when determining whether or not to forward messages. These parameters compel devices located beyond distance d from the initial sender of a message to forward messages with a probability p. Devices residing inside the scope defined by d gossip with probability 1, thereby preventing the gossip from potentially dying in conditions where a sender has relatively few neighbours. The impact of exploiting this optimisation technique on the number of transmitted messages and the message loss ratio have been investigated in the evaluation chapter of this thesis.

Simulating Radio Transmission Range Limitations

The algorithm shown in Figure 5.18 also includes a mechanism for simulating the limits of the physical radio transmission range of a device. Applications may specify a transmission radius that causes this simulator to discard all messages that have been received from devices beyond that range. This mechanism has proven to be valuable when evaluating prototypical STEAM application scenarios.

5.5 Exploiting Geographical Location Information

The STEAM event service has been designed to support collaborative applications in which application components can be either stationary or mobile and interact based on their geographical location. This implies that the STEAM middleware as well as the entities hosted by a particular mobile device are aware of their geographical location at any given time.

Sensing Location

STEAM comprises a location service that uses sensor data to compute the current geographical location of its mobile device and subsequently supplies this location information to the middleware and to the producers and consumers hosted by the device. The location service may collect data from a single sensor or from a range of sensors and use a data

fusion algorithm to compute its current location. Using data provided by multiple sensors may increase the accuracy of the location. The variety of available sensors may depend on the environment in which they are used, for example whether location data is provided indoors or outdoors, and on the service infrastructure that is available to them. For example, location systems, such as Bats [21] and Cricket [108], use sensors that rely on a previously deployed infrastructure when computing location data. These systems sense indoor locations by interacting with devices installed at known, fixed locations. A large range of sensors, including compass, speedometer, inertial tracker, GPS and differential GPS receivers, as well as ultrasonic sensors [21, 108], may be exploited for providing data to the location service. In order to accommodate outdoor applications, for example in the traffic management domain, STEAM exploits a version of the location service that uses a GPS satellite receiver to supply two-dimensional location information based on latitude and longitude coordinates.

Simulating Migration

In addition to providing location information based on sensor data, the location service can supply location information using a set of predefined coordinates. This allows applications to specify routes that describe the migration paths of certain entities, which are then used to simulate entity movements along these paths. For example, an application might define the route according to which an entity representing a vehicle travels. This simulation mode of the location service can be used for modelling the interactions of moving entities without having to provide for either an actual environment or a potentially vast quantity of equipment and has, especially in combination with the transmission range simulator, proven to be invaluable for the evaluation of prototypical STEAM applications. The combination of transmission range and migration simulators enables entities hosted by physically nearby devices to simulate migration as well as the potentially resulting loss of connectivity.

5.6 Summary

This chapter presented a prototypical implementation of the STEAM event model. The architecture of the STEAM event service incorporates the use of group communication and location-awareness in order to provide proximity-based event notification dissemination for mobile entities. Every mobile device hosting STEAM has identical capabilities that enable entities to interact through IEEE 802.11b-based ad hoc networks and provide a means for entities to announce and discover proximity-based events as well as to subscribe to and raise event notifications.

This STEAM event service allows applications to define event types as objects that comprise subject, content, and non-functional attributes. These objects are used to verify that raised event notifications conform to their types. STEAM maintains these types together with event delivery handlers and subject, content, and proximity filters in decentralised repositories.

The key enabling techniques used by the middleware include a distributed discovery service, a decentralised addressing scheme, and a location service that uses sensor data to compute the current geographical location of mobile devices and the entities hosted by them. The discovery service allows entities to announce and discover event type and proximity filter pairs that describe either stationary or mobile scopes and therefore, comprises two algorithms, one for disseminating event type and proximity filter pair announcements and another for handling such announcements upon receiving them. Both algorithms operate on the type repository of a particular mobile device and collaborate in order to maintain the relevant locally and remotely specified event type and proximity filter pairs. The addressing scheme enables mobile devices to recognise proximities of interest and to locally compute proximity group identifiers from event type and proximity information. This technique is based on using serialised descriptions of individual event type and proximity pairs, which are considered to be unique throughout a system, as the stimuli for a hashing algorithm generating group identifiers.

This chapter also presented the routing protocol that has been implemented to disseminate event notifications using either single-hop or multi-hop messages as well as the gossip-based optimisation mechanism used to reduce the number of forwarded multi-hop messages.

CHAPTER 6: EVALUATION

This chapter evaluates the approach to supporting collaborative applications that is being proposed in this thesis. We have selected a number of collaborative application scenarios from a range of domains, including traffic management, augmented reality, and emergency services, in order to reflect various application behaviours and to demonstrate how the components that comprise these collaborative applications can be interconnected in inherently distributed topologies through wireless communication and ad hoc networks. Specifically, we present a number of evaluation experiments, which have been conducted using these scenarios, to demonstrate how the objectives of this thesis have been met with respect to event notification filtering precision and system scalability, and to show the cost of event notification dissemination, proximity discovery, and event notification processing. The specific goal of each of the experiments along with the relevant configuration parameters are outlined in detail in their respective sections below.

All evaluation experiments presented in this thesis have been conducted by implementing the selected scenarios as prototypical applications. Each of the consuming and producing entities that comprise these applications is represented by an independent STEAM instance and interacts with other entities using a real ad hoc network. The entities and their STEAM instances are hosted by a number of notebook computers running the Microsoft Windows XP operating system on a 1GHz Intel Pentium III processor. Each machine is equipped with a Lucent Orinoco Gold WiFi PCMCIA card with a channel capacity of 11 Mbit/s providing the ad hoc network connection for inter entity communication. One or more entities may reside on a notebook computer. Entity locations and mobility are simulated throughout these experiments using the location service of their respective STEAM instances while the hosting notebook computers were placed within ad hoc communication reach of each other. The radio transmission range T_R for each machine was simulated to be T_R = 200 meters. This ensures that an entity discards all communication messages received from entities located beyond T_R, even though the distance between the physical locations of their host machines is less than T_R. Multiple runs were conducted for each experiment and the data collected was averaged over those runs.

6.1 Disseminating Event Notifications

The main objective of this experiment is to evaluate how exploiting proximity limits event notification forwarding and consequently the cost of disseminating event notifications. The primary measurement of interest is an abstract quantity we refer to as cost. We assign a relative cost to the dissemination of a single event notification. Cost describes the number of messages required when propagating an event notification from a producer to the consumers residing within its radio transmission reach and the forwarding of this message to consumers beyond this range. Hence, cost depends on the number of consumers residing within a particular proximity and provides a qualitative indication of the bandwidth required for event notification dissemination. For example, the cost of a producer disseminating an event notification to three subscribers, each of which forwarding the message that describes the event notification once, is described as 4; one message sent by the producer and 3 messages forwarded by the subscribers. This allows us to measure the effect of different application behaviours on this cost independently of the actual content of an event notification while varying a number of application parameters. These parameters include the migration speed of the entities, the range of the proximity within which event notifications are disseminated, the number of subscribers, and the distribution of these subscribers.

6.1.1 The Application Scenarios

We have selected a number of scenarios from various application domains for this experiment. These scenarios differ in the distribution of their respective consumers relative to a producer and as a result, in the ad hoc network topologies through which event notifications are disseminated. Table 6.1 summarises the chosen application scenarios and outlines the type of circular shaped proximity used.

Scenario	Application Domain	Description	Proximity Type
(A_1) (A_2)	Traffic Management	This scenario includes a broken down car disseminating accident warnings to vehicles within its vicinity. Figure 6.1 depicts that this scenario is set on a two way road. A producer acting as the broken down car propagates event notifications to consumers representing the vehicles. These vehicles are randomly distributed on the lanes of the road. Scenario (A_1) uses a static network topology (defined by the vehicle distribution) to measure the effect of proximity range and the number of subscribers on event dissemination cost. Scenario (A_2) focuses on the effect of subscriber speed and thus, uses a dynamically changing network topology that reflects vehicle movements. The vehicles start their journeys at their respective random locations, drive along the lanes, and turn upon reaching the end of the part of the road specified by the application.	Absolute

Scenario	Application Domain	Description	Proximity Type
(B_1) (B_2)	Traffic Management	This scenario includes an ambulance disseminating event notifications to nearby vehicles for them to yield the right of way. As shown in Figure 6.1, this scenario is set similarly to the previous scenario on a two way road and comprises an ambulance entity propagating its event notifications to a number of randomly distributed consumers representing vehicles. Scenarios (B_1) and (B_2), like scenarios (A_1) and (A_2), concentrate on static and dynamic network topologies respectively. In contrast to scenario (A_2), which comprises a static producer and moving consumers, scenario (B_2) involves the ambulance moving along the road propagating event notifications to stopped vehicles. The other main difference to the previous scenario is the type of the proximity used.	Relative
(C)	Traffic Management	This scenario includes a traffic light propagating its status to the vehicles passing through an intersection. As outlined in Figure 6.1, these vehicles are randomly distributed on the lanes approaching a four way intersection. This scenario, as well as all subsequent scenarios, consider static network topologies only.	Absolute
(D)	Augmented Reality	This scenario includes an augmented reality game in which a particular player informs other players of its game status. Figure 6.1 illustrates that these players can reside anywhere inside the game space, which is described by the proximity, and hence, their locations have been determined randomly.	Absolute
(E)	Emergency Services	This scenario includes a search and rescue operation comprising a coordinator directing a group of distributed searchers, each responsible for examining an equal part of the search area. Figure 6.1 depicts these searchers being homogeneously distributed in the search area defined by the proximity.	Absolute

Table 6.1. Description of the application scenarios.

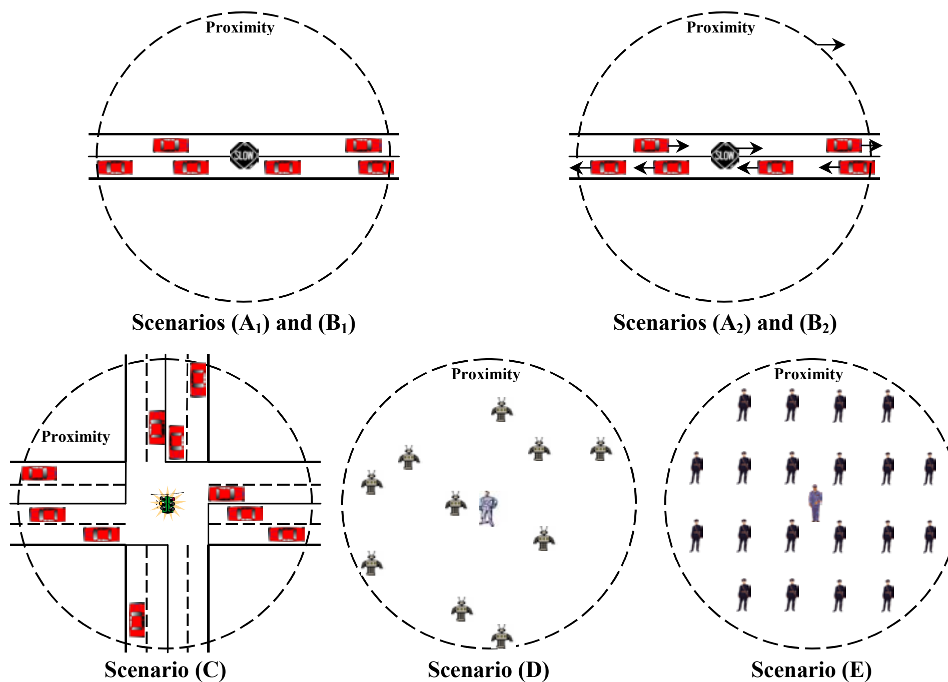


Figure 6.1. Application scenario overview.

6.1.2 The Experiment

The experiment comprises four notebook computers placed on the sides of a 5 by 5 meters square communicating through wireless ad hoc connections. One machine hosts the event producing entity and the remaining three host the consumers. Each of these machines accommodate an equal share of the consumers.

The cost of event dissemination has been evaluated for each of the application scenarios. The behaviour of each scenario involves the basic service requests of the producer announcing the subject of its events together with the associated proximity before publishing event notifications and consumers subscribing to them. All consumers are assumed to be interested in these event notifications and thus, do not filter the event notification content.

Parameters

The measurements have been conducted as a function of a number of application parameters.

Saturation: Each scenario involves a geographical area of circular shape, defined by a radius of 700 meters, surrounding its producer. All consumers reside within this area and their

number defines the saturation of the area. Saturations range from sparse populations, comprising 60 consumers, to dense populations consisting of 240 consumers. These saturations reflect congestion in the traffic management scenarios and result in average distances between vehicle locations varying from 47 meters to 12 meters in (A) and (B), and from 187 meters to 47 meters in (C). They define the number of payers residing in (D)'s game space of 1.5 km^2 and the search area for each searcher in (E) to range from 160 by 160 meters to 80 by 80 meters.

Proximity: Proximity defines the circular shaped, geographical area surrounding a producer within which event notifications are disseminated. In other words, proximity defines the set of consumers residing inside the area of interested to which event notifications are disseminated. The radius of this proximity P_R varies from 100 meters to 700 meters. These ranges have been selected to include both proximities in which all consumers can be reached using a single-hop radio transmission ($P_R <= T_R$) and proximities that require multi-hop routing ($P_R > T_R$). The largest proximity covers the whole scenario area enabling event notification propagation to all consumers. In scenarios (A), (B), and (C), the variations of the proximity range may reflect different road conditions causing changes to the speed limit. In scenarios (D) and (E), these ranges may reflect different game spaces and search areas respectively.

Speed: Producers and consumers may migrate according to predefined routes with a certain speed. The speed chosen for scenarios (A_2) and (B_2), which evaluate event notification dissemination cost as a function of mobility, ranges from 10 miles per hour to 70 miles per hour. In scenarios (A_2), this selection reflects vehicles approaching an accident site on a road with varying speed limits and road conditions (congestion, weather, time of the day) that may cause them to drive even slower than the actual speed limit. Similarly, the varying speed in scenario (B_2) indicates situations ranging from a slowly moving ambulance navigating through heavy traffic to an ambulance driving at high speed.

Protocols

The main objective of this experiment is to record the cost of event notification dissemination using application scenarios (A) to (E) while exploiting STEAM's proximity-based multicast protocol. An additional goal involves evaluating the effect of the proposed gossip-based optimisation technique on this cost. We have repeated the measurements conducted on the scenarios based on static ad hoc network topologies using the previously described configuration, while applying the gossip parameters G(d=200.0, p=0.8). These parameters compel entities outside the single-hop distance of d = 200.0 meters to retransmit messages

with a probability of p = 0.8. This combination of retransmission distance and relatively high gossip probability [107] have been chosen with the intention of preventing message loss. Conducting this experiment with a single set of gossip parameters suffices to evaluate the effect of this optimisation technique on the cost. Discovering the ideal gossip parameters for the respective application scenarios is considered outside the scope of this thesis.

Proximity Shape

We have chosen to exploit circular shaped proximities in this experiment since they provide a natural means to specify an area surrounding a single producer. In addition, they enable entities to use a simple distance calculation when determining the status of their proximity membership. Exploiting other proximity shapes, such as rectangles, affects the latency for determining whether or not an entity is located inside a particular proximity, but does not influence the cost evaluated in this experiment.

Latency of Event Notification Dissemination

This experiment is concerned with the evaluation of event notification dissemination cost outlining an indication of the bandwidth required. It does not consider the latency of event notification dissemination as we argue that latency might be of lesser interest in a system based on asynchronous, best-effort communication. Moreover, this latency depends directly on the topology of the underlying ad hoc network in terms of the physical location of the participating nodes. Hence, conducting meaningful latency measurements using our scenarios requires the deployment of a significant number of machines hosting the entities that comprise the applications.

Overhead

The overhead caused by proximity discovery is excluded in this experiment and is evaluated separately in section 6.2. This overhead reflects the cost of proximity discovery for a certain time period and as such cannot be mapped directly to the dissemination cost of a single event notification. Proximity discovery does not influence the results of the measurements conducted on the scenarios based on static network topologies provided proximities are being discover prior to event notification dissemination. Subsequent updates are not required. In scenarios based on dynamic network topologies, proximity discovery is configured to propagate updates prior to the dissemination of each event notification in order to accurately reflect the actual topology.

Coverage

Coverage can be defined as the geographical area to which a particular producer can send event notifications using either single-hop or multi-hop communication. A proximity is said to be covered if the associated set of subscribers is a subset of the entitles to which a specific producer can propagate event notifications. Network partitions may occur in cases where an area defined by a proximity is not covered by a specific producer. In general, they are more likely to occur in application scenarios exploiting sparsely populated proximities. The saturations in this experiment have been chosen with the intention of preventing message loss due to coverage limitations. In order to verify this, the message loss ratio is recorded for all measurements.

6.1.3 Results and Analysis

This section summarises the results of the conducted measurements and discusses the findings.

Cost of Event Dissemination

Figure 6.2, Figure 6.3, and Figure 6.4 illustrate the cost of event notification dissemination as a function of proximity range for the static ad hoc network topologies described by application scenarios (A_1), (B_1), and (C) to (E).

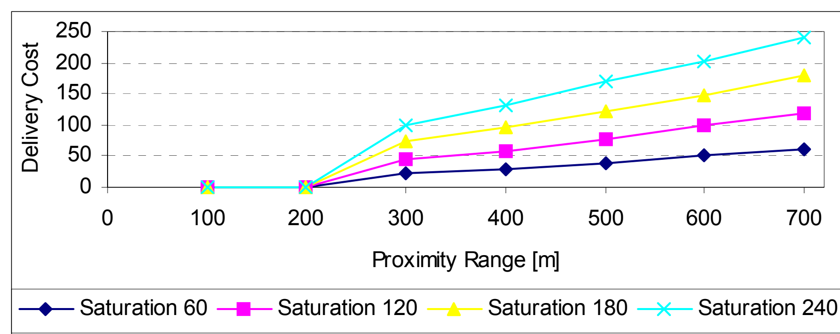


Figure 6.2. Cost of event notification dissemination in scenario (A_1) as a function of proximity range.

These figures show a significant difference when comparing event dissemination cost within the single-hop reach to the cost beyond this range. The cost of disseminating event notifications inside the single-hop reach is low and independent of both proximity range and

saturation as a single message suffices to reach all subscribers. Beyond the singe-hop range, all results show similar tendencies of increasing costs with expanding proximities and rising saturations as every subscriber residing inside a certain proximity forwards messages. Increasing a proximity causes an rising number of subscribers to forward messages in order to cover a larger geographical area. In essence, these figures illustrate how proximities bound event notification dissemination cost by bounding the number of subscribers that forward a certain message. Other approaches [106] characteristically use mechanism based on hop distance from source to destination to control flooding in ad hoc networks.

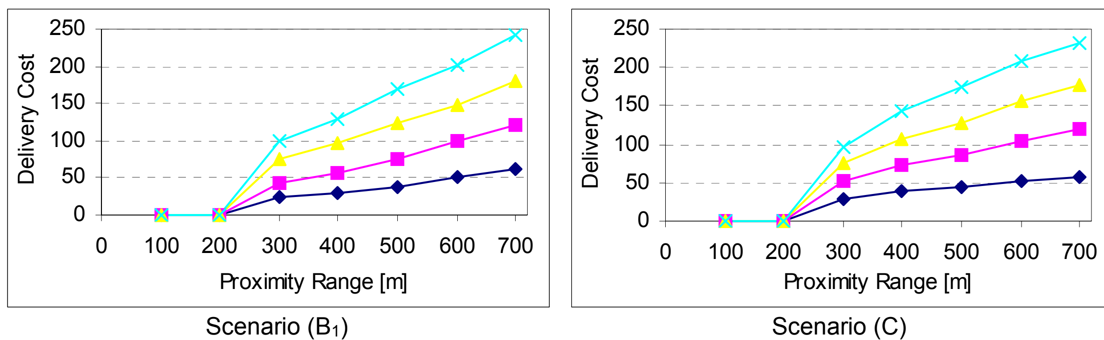


Figure 6.3. Cost of event notification dissemination in scenarios (B_1) and (C) as a function of proximity range.

These figures also illustrate that exploiting proximity for defining propagation ranges enables STEAM to transparently select the appropriate protocol when disseminating event notifications. STEAM uses its cost efficient single-hop protocol for raising event notifications in proximities that are covered by the producer's T_R and employs the multi-hop version only when transmitting messages beyond T_R. Other middleware platforms typically use either a single-hop protocol with propagation range limitations or a more expensive multi-hop protocol for both short and long range event notifications dissemination.

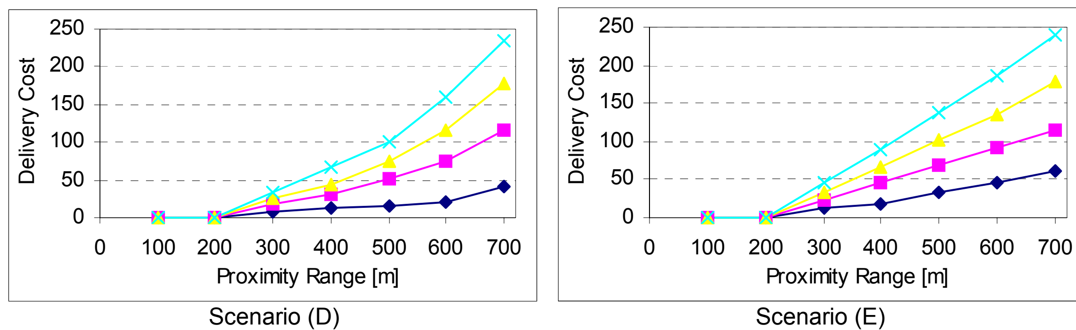


Figure 6.4. Cost of event notification dissemination in scenarios (D) and (E) as a function of proximity range.

Furthermore, the results recorded for scenario (A_1) and (B_1) are virtually identical demonstrating that, given a similar configuration, the cost for disseminating event notifications in absolute and relative proximities are comparable.

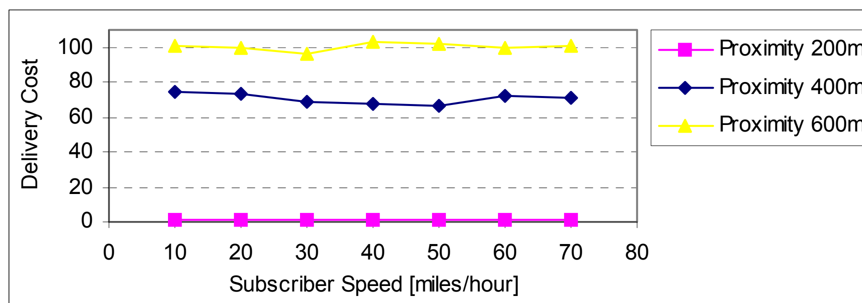


Figure 6.5. Cost of event notification dissemination in scenario (A_2) for a saturation of 120 as a function of subscriber speed.

Figure 6.5 and Figure 6.6 depict the delivery costs recorded for scenarios (A_2) and (B_2) respectively as a function of migration speed for a subset of the proximities previously used. These proximities have been chosen to include both single-hop and multi-hop transmissions ranges. Essentially, both figures show that these costs, similar to the costs in static network topologies, are low within single-hop reach and increase with expanding proximities beyond single-hop range. Significantly, they illustrate that cost does neither depend on the speed of subscribers nor on the speed of the producer. This is due to the fact that STEAM exploits proximities to control multicast-based flooding and consequently does not introduce extra overhead for maintaining routing information that needs to be updated more frequently with increasing migration speed. The study of flooding-based multicast protocols for ad hoc networks presented by Lee et al. [103] presents similar conclusions and hence, argues that

neither the number of transmitted messages nor the associated delivery ratio is a function of the speed of the communicating entities.

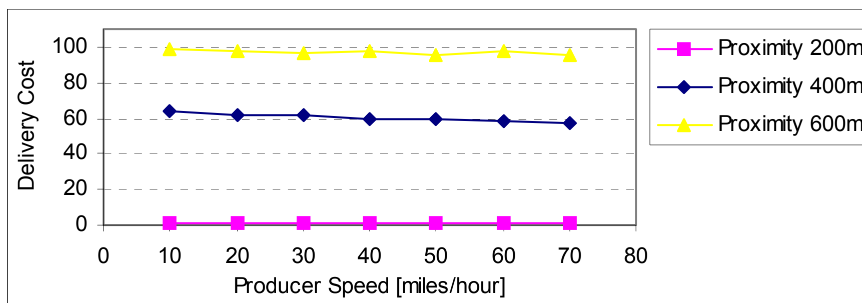


Figure 6.6. Cost of event notification dissemination in scenario (B_2) for a saturation of 120 as a function of producer speed.

Cost of Event Dissemination Using Gossiping

Figure 6.7 illustrates the reduction in dissemination cost when applying a gossip-based optimisation technique to our application scenarios. The results shown represent the cost reduction averaged over the data found for saturations ranging from 120 to 240 for each scenario. As discussed below, cost reduction measured for saturations 60 was found to be rendered meaningless due to message losses and have thus been excluded. In essence, Figure 6.7 demonstrates that a significant dissemination cost reduction can be achieved by using optimisation techniques provided the saturation is sufficiently high to prevent message loss. For the given application scenarios and the chosen retransmission probability, saturations of 120 or more were found to fulfil this requirement.

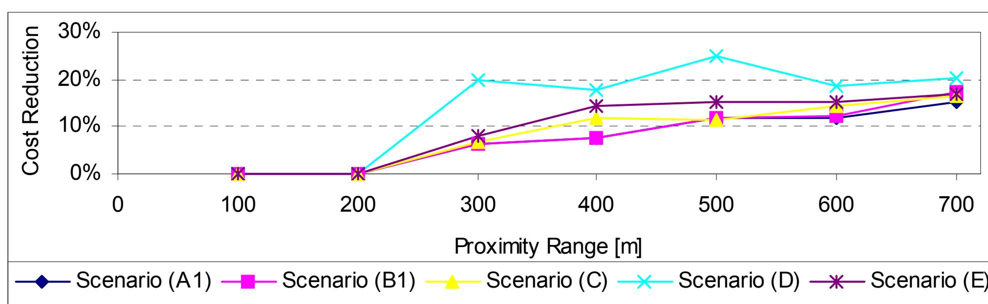


Figure 6.7. Event notification dissemination cost reduction due to gossiping.

Message Loss Ratio

The message loss ratio was found to be negligible in almost all measurements. Only the measurements conducted with the smallest saturations (60) and gossiping consumers in scenarios (C), (D), and (E) caused the ad hoc network to partition. As a result, a significant number of consumers were excluded from the ad hoc network and did not receive any messages. As summarised in Figure 6.8, message loss ratios of up to 30% are recorded for proximity ranges above 300 meters. An optimisation technique based on a gossiping probability that depends on the number of node neighbours [107] might be better suited for these scenarios. Scenarios (A_1) and (B_1) were not affected as their consumers are distributed in a relatively small subsection of the proximity area.

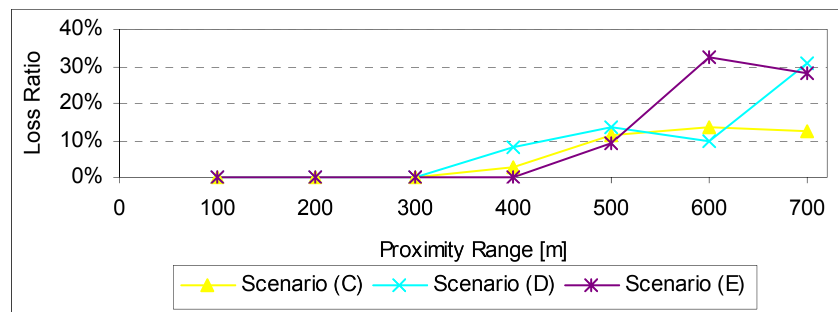


Figure 6.8. Fraction of gossiping consumers losing event notifications in scenarios with a saturation of 60.

6.2 Discovering Proximities

The main objective of this experiment is to examine the cost of announcing event types and associated proximities. In other words, this experiment evaluates the overhead of discovering event notifications of interest. Similar to the previous experiment, the primary measurement of interest is an abstract quantity we refer to as cost. We assign a relative cost to announcing and discovering a proximity associated with a specific event type. Cost describes the number of messages required when propagating an announcement message from a producer to the consumers residing within its radio transmission reach and the forwarding of this message to consumers beyond this range. Cost depends on the number of consumers residing within the proximity that is announced and on the consumer discovery range, which is defined individually by consumers allowing them to describe their scope of interest. Hence, cost provides an qualitative indication of the bandwidth required for announcement dissemination.

Since announcement messages are propagated using periodic beacons, costs reflect the discovery overhead related to a certain time interval. We measure the effect of different application behaviours on this cost as a function of the consumer discovery range.

The results of this experiment outline the main communication overhead imposed by the STEAM middleware. Other basic service requests in STEAM, such as subscribe and unsubscribe, do not result in additional inter-entity interaction.

6.2.1 The Application Scenarios

The results of the previously presented experiment outline that neither proximity type nor entity speed affects the cost of propagating messages in STEAM. We have therefore chosen to use a subset of the application scenarios presented in section 6.1 in this experiment, namely scenarios (C) and (D), expecting them to characteristically manifest the cost for discovering proximities. These scenarios feature different application behaviour which is reflected by their ad hoc network topologies. Specifically, scenario (C) features application behaviour where entities randomly populate a subsection of a given proximity whereas the entities in scenario (D) reside at arbitrary locations anywhere inside a certain proximity.

6.2.2 The Experiment

The experiment has been set up using the configuration previously described in section 6.1 and proximity discovery cost has been measured for both application scenarios. The behaviour of each scenario involves the producer announcing the type of its event notifications together with the associated proximity and consumers subscribing to this event type.

Parameters

The measurements have been conducted as a function of a number of application parameters.

Saturation: The previously described saturations ranging from 60 to 240 are used in this experiment.

Proximity: The proximity ranges have been chosen to include P_R's of 200 meters and 400 meters. The former allows STEAM to propagate announcement messages using single-hop

transmissions whereas the latter requires multi-hop transmissions. Thus, these ranges have been selected in order to evaluate both transmission modes.

Discovery range: The radius of the circular shaped discovery range, describing the consumer scope of interest, is varied from 50 meters to 600 meters in order to cover the two proximities for which the measurements are conducted. Short ranges are chosen by consumers that are only interested in discovering proximities in their immediate vicinity whereas larger ranges enable them to discover proximities in a wider area. For example, vehicles in scenario (C) might wish to use different discovery ranges depending on their maximum speed; an agricultural vehicle may be content with a small discovery range whereas a police car may require a substantially larger range. Players in scenario (D) might vary their discovery ranges depending on whether they migrate in an urban or an open game space.

6.2.3 Results and Analysis

As previously described, the measurement are conducted using various saturations for each of the scenarios. The results shown have been averaged over the data collected for these saturations.

Discovery Cost

Figure 6.9 shows the cost of discovering proximities as a function of the discovery range. The cost for discovering a proximity in single-hop scenarios (SH Scenario (X)) is constant and, compared to multi-hop scenarios (MH Scenario (X)), very low. Moreover, this cost is independent of the discovery range, which can be explained by the fact that message propagation is limited by the proximity range.

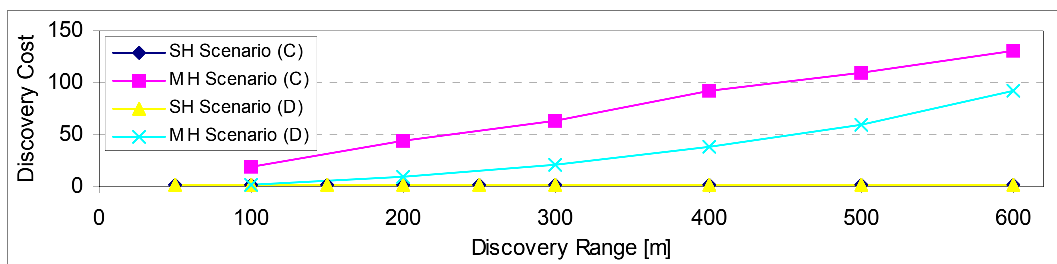


Figure 6.9. Cost of proximity discovery in scenarios (C) and (D) as a function of the discovery range.

The cost of proximity discovery in multi-hop scenarios grows linearly with increasing discovery range since messages are being forwarded to cover a larger discovery scope. However, this cost can be limited by choosing a discovery range that is appropriately small for a particular application.

Coverage

Figure 6.10 illustrates the discovery coverage for both scenarios as a function of the discovery range. This discovery coverage defines the ratio of the number of entities discovering a specific proximity to the number of entities residing inside this proximity. A discovery coverage of less than 100% indicates entities residing inside an undiscovered proximity whereas a discovery coverage of more than 100% signifies that some entities located beyond a proximity range discover the proximity. Discovery coverage is important from an application programmers perspective as entities unaware of a certain proximity will neither be able to join nor to receive disseminated event notifications.

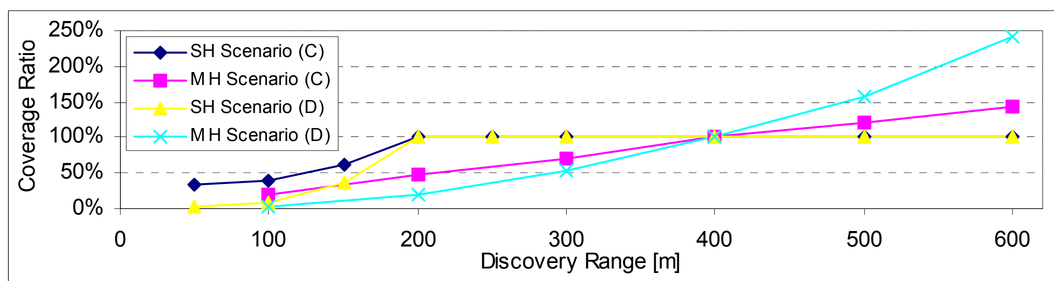


Figure 6.10. Discovery coverage ratio in scenarios (C) and (D) as a function of the discovery range.

As shown in Figure 6.10, coverage in single-hop scenarios remains at 100% once the discovery range exceeds the proximity range. Entities residing outside the proximity will not receive any announcement messages regardless of their discovery range. This implies that discovery ranges larger than proximity ranges do not impose additional overhead. In contrast, coverage in multi-hop scenarios increases beyond 100% for discovery ranges exceeding proximity ranges adding extra overhead. Discovery ranges that match proximity ranges are ideal in that they do not add extra overhead while enabling all entitles residing within the proximity to receive announcement messages. However, such an ideal range may not exist for some application since multiple, potentially different proximity ranges may be associated with certain event types.

In essence, discovery cost can be reduced by decreasing the discovery range, but discovery ranges that are smaller than proximity ranges compromise discovery coverage and consequently may lead to event notification loss.

6.3 Event Notification Filtering Precision

This section presents our evaluation of STEAM's event notification filtering engine, which demonstrates that combining event notification filters, each applied to a different event notification attribute, is beneficial as it increases the filtering precision. A prototypical application scenario from the traffic management domain has been implemented for this experiment. The scenario simulates the interaction between vehicles passing through an intersection and the intersection's traffic light disseminating its light status. The scenario is modelled according to the intersection of North Circular Road (NCR) and Prussia Street (PST) located in Dublin's inner city. It is based on real data, which has been provided by Dublin City Council [109], describing vehicle movements and light status at the intersection over a period of 24 hours.

6.3.1 The Intersection Scenario

Figure 6.11 illustrates the intersection and outlines how the traffic flow can be broken up into two distinct phases. The intersection comprises two approaches; approach one describes the traffic flows arriving from east and west whereas approach two describes the traffic flows arriving from north and south. Approach one comprises three lanes and approach two comprises of four lanes. The traffic light for both approaches is considered to be located in the centre of the intersection at the stated latitude and longitude coordinate.

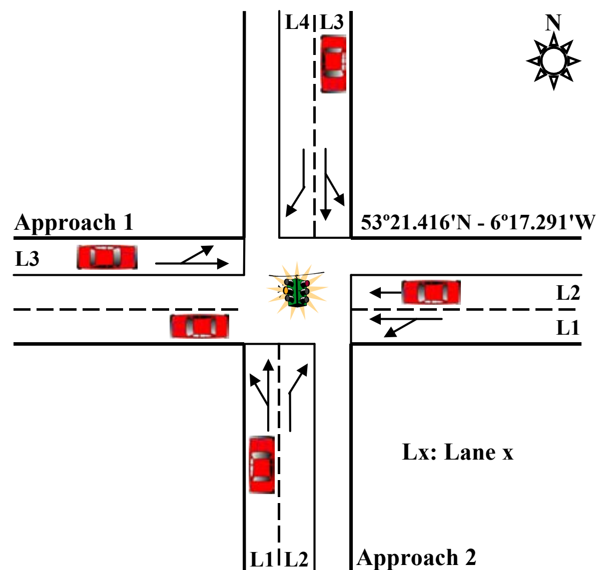


Figure 6.11. The North Circular Road and Prussia Street intersection.

The Data

The intersection data provided by Dublin City Council's traffic management system was acquired over a period of 24 hours starting on the 3rd of December 2002 at 6 pm. As outlined in Table 6.2, the data consists of a sequence of data records, each describing a cycle duration and the number of vehicles passing through the intersection on each individual lane during the cycle.

Record	Time	Cycle [s]	Number of Vehicles						
			Approach 1 (NCR)			Approach 2 (PST)			
			Lane 1	Lane 2	Lane 3	Lane 1	Lane 2	Lane 3	Lane 4
1	18:01	118	12	5	0	10	5	0	0
2	18:03	120	8	4	0	7	2	0	0
3	18:05	120	6	5	0	12	0	0	0
…									

Table 6.2. The first three records of the intersection data.

The example phase scheme of Figure 6.12 shows how the cycle time defines the duration for two approaches to complete their respective phases. Each phase consists of a sequence of light changes. The proportion of the cycle length that is assigned to one particular phase is

called the split. The split between the phase of approach one and two is constant at a ratio of 45% to 55%.

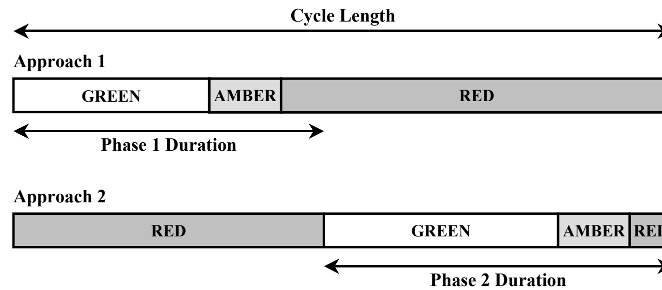


Figure 6.12. Example phase scheme for a two-approach intersection.

6.3.2 The Experiment

The experiment includes three notebook computers placed 5 meters apart communicating through wireless ad hoc connections. One machine hosts the traffic light, acting as a STEAM producer, and the other two accommodate the vehicles, represented as STEAM consumers. The vehicles are split between the two consumer machines according to the approach on which they travel. Each of these vehicles is connected to a separate STEAM instance with an independent location service simulating the vehicle's route.

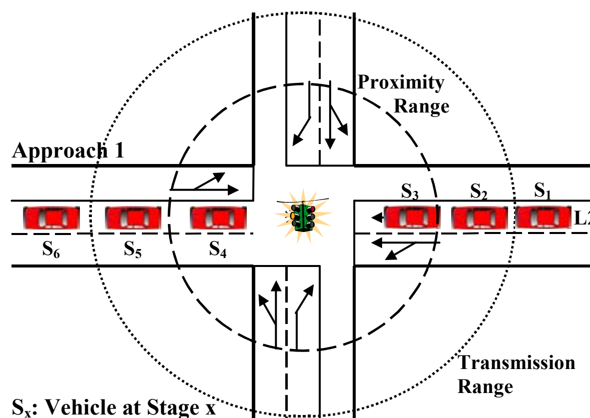


Figure 6.13. Modelling the intersection.

The traffic light raises an event of type "Traffic Light" every second for each approach. Each of these events describes the light status and contains the name of the approach as well as

the location of the light. Vehicles approach the intersection in their respective lanes at an average speed of 25 miles per hour (the intersection is located in a 30 miles per hour zone). Each vehicle follows a pre-defined route according to its approach lane. Figure 6.13 depicts an example route of a vehicle in lane two of approach one. The vehicle is shown at stages S_1 to S_6 of its journey through the intersection. The available intersection data does not describe the behaviour of an approaching vehicle in terms of queuing; it only indicates the number of vehicles passing the intersection during a green light sequence. Hence, vehicles are modelled to reflect this behaviour arriving at the intersection in time to pass the light during a green light sequence. Table 6.3 summarises the number of events raised for each approach and the total number of vehicles passing through the intersection on each individual lane.

	Approach 1 (NCR)			**Approach 2 (PST)**			
	Lane 1	**Lane 2**	**Lane 3**	**Lane 1**	**Lane 2**	**Lane 3**	**Lane 4**
Number of Events Propagated	84373			84373			
Number of Vehicles	6652	3320	3728	3038	1383	2802	1135

Table 6.3. The configuration of the experiment.

Configuration

This experiment comprises multiple runs, each run exploiting a different combination of event notification filters. Each of these runs simulates the intersection interactions using the same stimuli while recording the number of event notifications delivered to the vehicles. Table 6.4 outlines the specific combinations of event filters used in the runs. The subject filter allows vehicles to subscribe to “Traffic Light” events and the content filter matches events on behalf of vehicles moving towards the traffic light on a particular approach. As shown in Figure 6.13, the absolute proximity filter defines the radius of the area of interest surrounding the traffic light. The radius has been set to 40 meters to allow for vehicle braking distance (16 meters) and update rate (once per second) of the location service simulating vehicle movements. This radius guarantees that an approaching vehicle receives at least two events before having to decide whether or not to stop at the light. In general, omitted filters do not affect event dissemination, events are simply passed on. The traffic light’s wireless transmitter limits the communication range of event dissemination in case of an omitted proximity filter. We assume the radio transmission range in the modelled urban environment to be 200 meters and hence, events will be ignored by vehicles travelling beyond this transmission reach. For example, vehicles in run (A), in which all filters are omitted, will receive all disseminated

events once they are inside the transmission range of the traffic light. The vehicle shown in Figure 6.13 would receive all events propagated at stages S_2 to S_5 of its journey.

Run	Subject Filter	Content Filter	Proximity Filter
(A)	None	None	None
(B)	"Traffic Light"	None	None
(C)	"Traffic Light"	Approach, Towards Light	None
(D)	"Traffic Light"	None	Proximity Range
(E)	"Traffic Light"	Approach, Towards Light	Proximity Range

Table 6.4. The configuration of the runs.

In essence, the traffic light announces its events and the associated proximity as required by the specific run and subsequently disseminates the light status for each approach. Vehicles specify the required subject and content filters before commencing their journeys through the radio transmission range of the intersection's traffic light. Depending on their location and the combination of filters used in a particular run, vehicles will receive and deliver subsets of the generated events.

Vehicle Speed

As previously described, all vehicles in this experiment approach the intersection at a given averaged speed of 25 miles per hour. Vehicles, which might travel at lower average speeds, for example due to congestion, or indeed speeding vehicles may receive different numbers of events compared to the results recorded in this experiment. Slow vehicles may receive more events and speeding vehicles may receive fewer events. Nevertheless, measurements on such vehicles might still indicate relative decreases similar to the results shown below as the ratios of overall, transmission range, and proximity travel times remain unchanged. These ratios and consequently the relative decreases might be affected for vehicles travelling at drastically varying speeds, for example when slowing down in order to avoid collision with pedestrians or cyclists. However, we expect similar experimental conclusions, with respect to event notification filtering precision.

6.3.3 Results and Analysis

Figure 6.14 and Figure 6.15 summarise the number of event notifications vehicles deliver to the application in each of the runs (A) to (E). Figure 6.14 illustrates the overall number of

event notifications delivered on each of the lanes of the intersection. These quantities depend on the number of vehicles passing through a particular lane and show the traffic flow being heaviest on lane 1 of the NCR approach. Figure 6.15 shows the average number of event notifications delivered by an individual vehicle, which is similar for each lane, and also shows substantial reduction in delivered events when applying content or proximity filters.

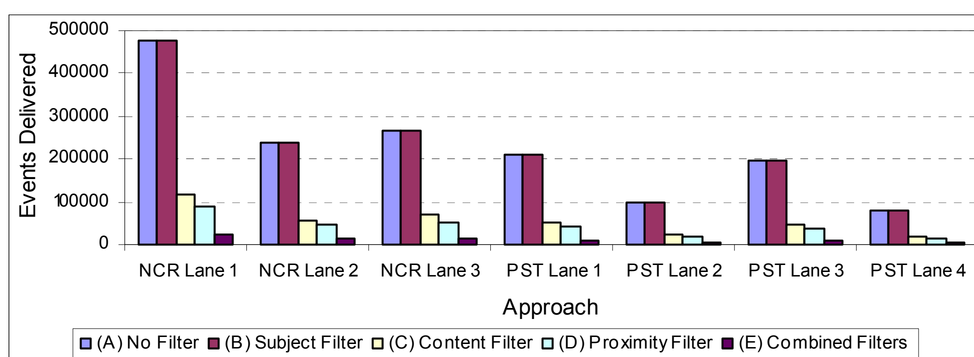


Figure 6.14. The number of event notifications delivered on each of the lanes.

Figure 6.16 demonstrates the precision of event filtering by comparing the results of runs (B) to (E) with the result of run (A) outlining the relative decrease in events delivered. The results of runs (A) and (B) are virtually identical and thus no decrease was measured. This is due to the fact that all events have the same subject in all runs. Events of an unrelated subject would have been delivered in (A) but not in (B). We argue that this is a straightforward concept and have therefore decided to exclude it from this experiment. The jitter between the results recorded for (A) and (B) is caused by the best effort semantics of our middleware.

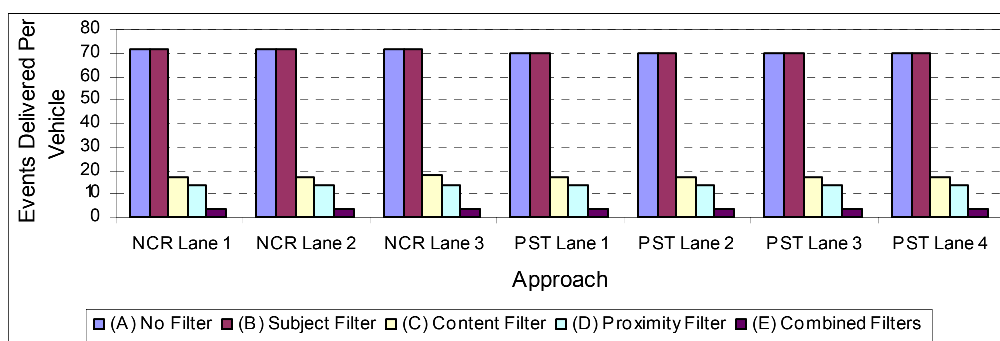


Figure 6.15. The average number of events delivered by individual vehicles on each lane.

Run (C) shows a relative decrease averaging at around 75%. The content filter matches events on the vehicle's approach, thus filtering approximately 50% of all disseminated events, and passes events only if a vehicle is moving towards the traffic light discarding 50% of the remaining events. The relative decrease found in run (D) averages at around 80%. The proximity filter bounds the propagation range preventing events from being delivered to vehicles travelling outside the proximity area. The ratio of transmission range to proximity range, which is 200 meters to 40 meters, accounts for the decrease found. The final run (E) measured a substantial relative decrease of approximately 95%. This is hardly surprising considering a combination of filters has been applied. In fact, this result can be explained by combining the decreases found in (C) and (D). Significantly, this run delivers the exact subset of event notifications in which this application is interested, discarding unwanted events.

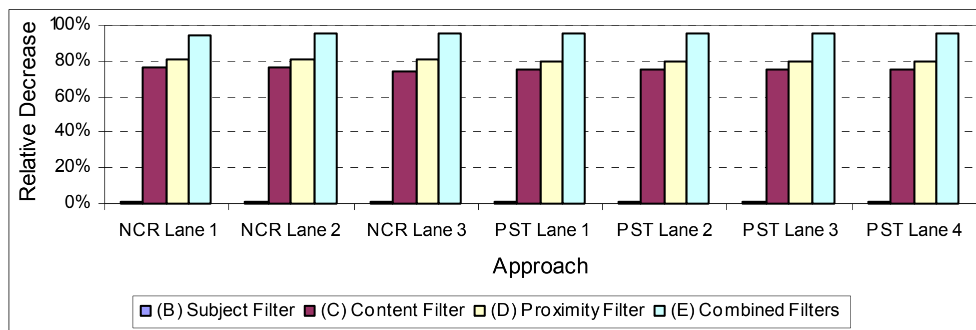


Figure 6.16. Precision of event notification filtering for various filter combinations.

6.4 Raising and Delivering Event Notifications

This experiment illustrates the behaviour of producers and consumers in terms of performance when raising and delivering event notifications respectively. In particular, it focuses on measuring the effect of parameters that typically describe the scale of a system on the latency of retrieving and matching various types of event notification filter. Hence, the main objective of this experiment is to demonstrate that our approach to maintaining event notification filters limits the effect of varying system scale on the latency of producers raising and consumers delivering event notifications.

6.4.1 The Experiment

This experiment comprises a producer hosted by one notebook computer disseminating events to one or more consumers running on another notebook. Each entity is connected to an independent STEAM instance and the host machines are placed 3 meters apart communicating through a wireless ad hoc connection. The locations of these entities have been chosen to allow the producer to reach all consumers with a single-hop radio transmission and the radius of the circular proximity area has been set accordingly.

Measurements

Measurements were conducted to determine the latency for producers to raise and for consumers to deliver event notifications. Producer latency represents the time for an application to invoke the raise operation and thus, includes processing (filtering and marshalling) and sending of an event notification. In other words, the recorded latency embodies an indication of the throughput of a producer. The consumer latency specifies the time to process a received event notification and to pass it to the application invoking the delivery handler. These measurements were conducted by averaging the latency over 100 runs. In addition, the latencies for retrieving and matching subject filters, content filters, and proximity filters have been recorded by averaging their latencies over 100,000 runs.

Event Type and Content Filter

The producer announces and generates event notifications of subject “Performance”. The corresponding even type, which is outlined in Figure 6.17, specifies a set of parameters comprising each of the parameter types currently supported by STEAM.

```
const int numPar = 5;
S_EventParameterDeclaration* epd;
SP_dsEventType* et;

epd = new S_EventParameterDeclaration[numPar];
epd[0] = S_EventParameterDeclaration("Integer Name", S_INT);
epd[1] = S_EventParameterDeclaration("Time Name", S_TIM);
epd[2] = S_EventParameterDeclaration("Location Name", S_POS);
epd[3] = S_EventParameterDeclaration("String Name", S_STR);
epd[4] = S_EventParameterDeclaration("Double Name", S_DBL);
et = new SP_dsEventType("Performance", numPar, &epd, SP_SINGLE_HOP);
```

Figure 6.17. Definition of the “Performance” event type.

Consumers subscribe to these event notifications and employ the conjunctive content filter shown in Figure 6.18. The specified filter expression consists of a set of filter terms, one of which is applied to each of the event notification parameters. The actual values of these filter terms have been chosen to match all disseminated event notifications.

```
SC_ConjunctiveContentFilter* cf;

cf = new SC_ConjunctiveContentFilter();
cf->addTermINT(0, SC_GREATER, 0);
cf->addTermTIM(1, SC_GREATER, 0);
cf->addTermPOS(2, SC_POS_WITHIN_RANGE, 200.0);
cf->addTermSTR(3, SC_STR_EQUAL, "Some Text");
cf->addTermDBL(4, SC_GREATER, 0.0);
```

Figure 6.18. Definition of the conjunctive content filter applied to the disseminated event notifications.

Parameters

The measurements have been conducted as a function of a number of application parameters.

Announcements: In addition to announcing (and raising) “Performance” event notifications, the producer announces a number of other event types (and their proximities) simulating a scenario in which it handles a (large) system that comprises several event types. This typically imposes extra computational load on producers.

Subscribers: The producer disseminates its event notification and indeed its announcements to a number of interested consumers, which subsequently subscribe to these event types. Similar to adding announcements, adding subscribers simulates the effect of increasing system scale on producers.

Subscriptions: Consumers may subscribe to several event types causing them to maintain multiple subscriptions and content filters. This simulates the effect of increasing system scale on consumers.

6.4.2 Results and Analysis

The data collected for producer and consumer latencies have been summarised in this section. The latencies for producers raising and consumers delivering event notifications are

shown. The latencies for retrieving and matching event notification filters are depicted as parts of the overall producer and consumer latencies.

Latency of Producers Raising Event Notifications

Figure 6.19 depicts the measured producer-side latencies as a function of the number of announced event types and associated proximities and the number of subscribed consumers. The latency for raising an event of type “Performance” was found to be approximately 1.8 milliseconds, which is equivalent to a throughput of just above 550 event notifications per second. The latencies recorded for subject filter retrieval and matching (SF rtv + match) and proximity filter retrieval (PF retrieval) are 1.55 and 1.45 microseconds respectively. Marshalling and sending event notifications (mainly) accounts for the remaining latency (Other).

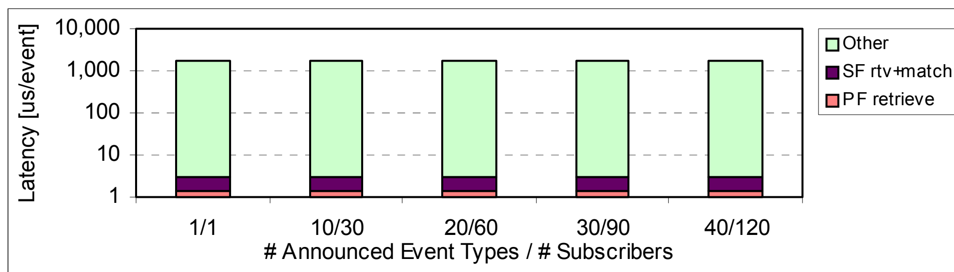


Figure 6.19. Latency of a producer raising an event notification as a function of the number of announced event notification types and the number of subscribers.

These results demonstrate that the latencies for raising event notifications and, in particular, for processing event notification filters are independent of the numbers of announced event types and subscribers.

Latency of Consumers Delivering Event Notifications

Figure 6.20 illustrates the recorded consumer side latencies as a function of the number of subscriptions to other event types maintained by the consumer. The latencies for retrieving and matching subject filters, content filters, and proximity filters, as well as the overall latency for delivering event notifications, are independent of the varying number of subscriptions. Marshalling and invocation of the application delivery handler (mainly) accounts for the remaining latency (Other).

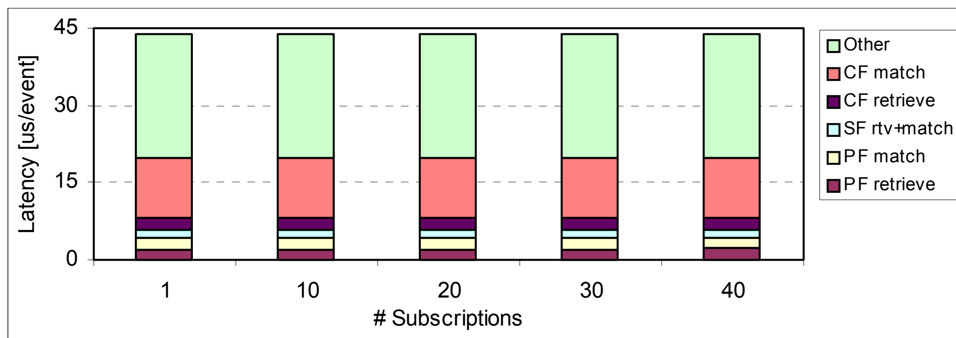


Figure 6.20. Latency of a consumer delivering an event notification as a function of the number of subscriptions to other event notification types.

Some of the latencies outlined in Figure 6.19 and Figure 6.20 are application specific. Content filter matching depends on the number of filter terms and on the type of the parameter to which these terms apply, proximity filter evaluation latency may vary with the geographical shape of the proximity area, and passing event notifications to an application depends on the delivery handler. Even though evaluating these application specific latencies is straightforward, we have decided to exclude such an evaluation as it represents a insignificant contribution in the context this thesis.

In conclusion, the measurements recorded in this experiment demonstrate that the latency for processing event notification filters, specifically their retrieval and matching, is independent of parameters defining the scale of a system.

6.5 Summary

This chapter presented four experiments evaluating various collaborative application scenarios using the STEAM middleware.

The first of these experiments shows the cost of disseminating event notifications for different application behaviours and various static and dynamic ad hoc network topologies. The experiment demonstrates how exploiting proximity limits this cost by bounding event notification forwarding. Using proximities for defining event propagation ranges allows STEAM to transparently select the appropriate protocol for disseminating event notifications. A cost effective single-hop protocol can be employed for disseminating event notifications within the radio transmission range of a producer whereas a more expensive multi-hop protocol has to be used for reaching entitles residing beyond the single-hop reach. The experiment also illustrates that event dissemination cost in dynamic ad hoc network

topologies is independent of both subscriber and producer migration speed. In addition, this experiment demonstrates that the cost of disseminating events in absolute and relative proximities are comparable and that optimisation techniques, for example based on gossiping, may be exploited to reduce dissemination cost in areas with a sufficiently high node saturation.

The second experiment shows the cost associated with announcing and discovering event types and proximities and thus, outlines the overhead of entities discovering event notifications of interest. Not surprisingly, the cost of disseminating announcement messages exhibited behaviour similar to the cost of disseminating event notifications in that a single-hop protocol suffices when discovering proximities within radio transmission range of the producer. Discovery cost is bounded by the proximity range *and* by the discovery range defined by consumers. This allows consumers to reduce discovery cost by decreasing their discovery range. However, this may compromise delivery coverage and consequently lead to event notification loss.

The third experiment evaluates STEAM's event notification filtering engine and demonstrates that combining event notification filters applied to different event notification attributes increases the filtering precision. This experiment models a specific intersection of Dublin's inner city and simulates interactions between vehicles moving through the intersection and the intersection's traffic light over a period of 24 hours based on real data provided by Dublin City Council. The relative decrease in the number of event notifications delivered to these vehicles has been measured for various filters. Although, applying either a content filter or a proximity filter results in a significant reduction in the number of delivered event notifications, the most significant decrease was found when combining subject filters, content filters, and proximity filters. Significantly, combining these filters resulted in the exact subset of event notifications of interest being delivered to this application and unwanted event notifications being discarded by the middleware.

The final experiment outlines the behaviour of producers and consumers in terms of performance when raising and delivering event notifications respectively while focusing on the effect of varying system scale on this behaviour. The measurements demonstrate that the latencies for processing event notifications and, in particular, for processing event notification filters are independent of application parameters that typically describe the scale of a system.

In addition, the results of these experiments demonstrate that various techniques employed by STEAM allow a system to easily cope with a large, dynamically changing population of entities distributed over a large geographical area. Exploiting proximities to bound event notification dissemination and their discovery allows an application to divide a large

geographical area into multiple, potentially independent sub-areas, each handling their event notifications locally. This enables STEAM to transparently select an appropriate protocol for disseminating messages and to limit message forwarding to the predefined area. Combining event notification filters that apply to different event notification attributes allows the middleware to discard unwanted event notifications and to process event notifications and especially their filters in a manner that is independent of the scale of a system and the number of deployed filters.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

This thesis presented the STEAM event-based middleware designed to support the highly mobile (and stationary) entities that comprise collaborative applications.

This chapter summarises the most significant achievements of the work described in this thesis and outlines its contribution to the state of the art. This thesis is then concluded with a discussion of related research issues that remain open for future work.

7.1 Achievements

The motivation for the work presented in this thesis arose from the observation that state of the art research in distributed event-based programming systems for wireless networks has mainly focussed on accommodating nomadic applications.

Nomadic applications are characterized by the fact that mobile entities make use of the wireless network primarily to connect to a fixed network infrastructure but may suffer periods of disconnection while moving between points of connectivity. As discussed in chapter 1 and chapter 2 of this thesis, the main goal of event-based middleware supporting nomadic applications has consequently been to handle disconnection while entities migrate from one designated access gateway to another. This implies that such middleware has focused on providing a means to cache and synchronise relevant information on behalf of a disconnected entity and to forward it via the new access gateway upon reconnection.

As described in chapter 2, event-based middleware traditionally employs logically centralised or intermediate components to implement key features and properties of the middleware. Application components may utilise centralised lookup and naming services to discover peers in order to communicate with them. Intermediate components may be used to route event notifications from producers to consumers and to apply event notification filters. Moreover, they may enforce non-functional attributes, such as event notification delivery order and priority. However, the central problem with this approach arises with increasing system scale

as such middleware components may become a liability due to availability and bandwidth limitations.

Centralised or intermediate middleware components are typically hosted by physical machines that are part of a designated service infrastructure in order to ensure that they are always accessible to application components. The disadvantage of exploiting such an infrastructure is that its installation and maintenance requires substantial resources while limiting communication between entities to the geographical areas in which the infrastructure has previously been made available.

A similar approach has been used by middleware for nomadic applications since designated middleware components, which typically implement the mechanisms for handling disconnection, can be hosted naturally by parts of the network infrastructure, for example by wireless access gateways as suggested by Huang et al. [6]. As discussed in chapter 2, this is also illustrated by SIENA and Elvin4 as both use intermediate proxy components, which are hosted by parts of the service infrastructure, for managing information on behalf of a moving entity. Likewise, JEDI and CEA use intermediate components for event notification dispatching that are part of the network infrastructure.

Such an approach is inadequate for event-based middleware supporting collaborative applications that can be characterised by the fact that mobile entities use the wireless network to interact with other mobile entities that have come together at some common location. Collaborative entities migrate within some area, establish associations with other entities dynamically, and typically group into formations of entities that have a common goal. Although these applications may use infrastructure networks, they will often use ad hoc networks since they are immediately deployable in arbitrary environments and support communication without the need for a separate infrastructure. Consequently, middleware accommodating collaborative applications can not rely on the presence of a designated service infrastructure.

It can be observed in the state of the art that other work does not attempt to support this style of mobile application in which collaboration is intrinsic when entities are in close proximity (see chapter 2), even though this application style can be useful in the ubiquitous and sentient computing domain allowing loosely coupled, inherently mobile entities to move together and apart over time.

The main challenge was to design event-based middleware that supports collaborative applications without the aid of a separate service infrastructure while avoiding centralised and intermediate components. A further challenge was to provide event filtering with high precision allowing an entity to use multiple functional and non-function criteria when

identifying event notifications of interest. The final challenge was to develop decentralised techniques that improve system scalability for applications composed of large numbers of interconnected mobile (and stationary) entities distributed over large geographical areas. This has been achieved. The resulting architecture, captured as an event model known as STEAM, has been presented in this thesis.

STEAM provides for unanticipated interactions among collaborative application components by allowing event producers to define geographical scopes that bound the areas in which their event notifications are relevant. Event consumers residing or indeed entering such areas can dynamically discover these proximities and subsequently establish logical connections to the associated producers. The connections between the entities residing in a particular proximity are then used by producers to disseminate their event notifications thereby allowing consumers to deliver events at the specific location where they are valid.

Such geographical scopes represent a natural way for mobile entities to identify events of interest and enable entities that have come together at a certain location to spontaneously discover and interact with each other. STEAM specifically addresses the needs of mobile applications by extending the concept of geographical scopes introduced by ECO (see chapter 2) with a notion of proximity in which areas of interest can be mobile as well as stationary.

The architecture of STEAM is inherently distributed and is based on an organisation with distributed collocated middleware [23]. The STEAM middleware is exclusively collocated with application components and depends neither on centralised or intermediate components nor on the presence of a designated service infrastructure.

STEAM's inherently distributed architecture implies that every STEAM instance offers identical capabilities to its application. Every physical machine hosting STEAM is capable of providing the same service to producers and consumers without accessing remote components. The design of STEAM facilitates this by supporting various distributed mechanisms to provide the desired middleware properties. Consequently, STEAM provides decentralised techniques for peer discovery based on beacons, for routing event notifications from producers to consumers without the aid of access points using multicast groups and a distributed addressing scheme, for enforcing non-functional attributes, such as event notification delivery order and priority, and for event filtering based on combining multiple, producer-side and consumer-side filters.

Event notifications can not depend on intermediate components applying event filters at a central location. STEAM therefore supports a distributed approach to filtering allowing event notifications to be filtered at both the producer side and the consumer side thereby enabling

applications to exploit the advantages of both filter locations. In particular, applying filters at the consumer side enables filtering on the context of event notifications, such as the geographical location (of consumers), that is not available at the producer side.

STEAM supports a range of event filters that may be applied to a variety of event notification attributes, including subject, content, and context, such as geographical location. As demonstrated in chapter 6, combining such event filters is beneficial to the precision of filtering allowing an entity to define the subset of event notifications in which it is interested using multiple criteria, such as meaning and location.

Many of the decentralised techniques we have discovered for supporting mobility naturally accommodate a dynamically changing population of mobile entities and as a result, help to improve the scalability of a system. STEAM's inherently distributed architecture avoids designated middleware components that may become communication bottlenecks with increasing system scale. The concept of proximity-based event dissemination bounds the geographical scope within which certain information is valid and thus, limits forwarding of event notification and configuration information which may lead to a reduction in the required communication and computational resources. Combining multiple event filters improves the filtering precision and consequently, reduces the number of potentially unwanted event notifications being propagated. Decentralised filtering also helps to improve the scalability of a system by distributing the computational load of filter matching as a small number of filters are typically evaluated on each specific machine.

Another of the decentralised techniques featured by STEAM is also worthy of mention. STEAM's distributed addressing scheme replaces the kind of centralised approach traditionally used for identifying peers of interest and therefore represents a key enabling mechanism to STEAM's inherently distributed architecture. This addressing scheme enables entities to recognise proximities of interest and to obtain the correct proximity group identifiers using device local rather than global knowledge based on a textual description of discovered event type and proximity pairs.

These concepts and techniques were realised in a prototype implementation of the STEAM event model. As described in chapter 6, a range of collaborative application scenarios were then selected to conduct a number of evaluation experiments. These experiments demonstrate how the objectives of this thesis were met with respect to event filtering precision and system scalability. They also show the cost of event dissemination, proximity discovery, and event notification processing.

A further contribution of this thesis is a taxonomy of distributed event-based programming systems. The taxonomy is structured as a hierarchy of the fundamental properties of a

distributed event-based programming system and can be used as a framework to describe an event system according to a variety of criteria including its event, organisation, and interaction models. The taxonomy was applied to our middleware as well as a selection of other event systems to compare their middleware properties.

7.2 Open Research Issues

As is always the case with research, there are some issues that remain open for possible future work. The event model presented in this thesis supports attributes that classify event notifications according to their timeliness requirements. The mechanism for enforcing these classes of events is likely to be affected by the dynamic nature of the mobile computing environment for which STEAM has been designed. The main area for future exploration therefore includes the issue of achieving timeliness and reliability for real-time event-based communication in ad hoc wireless networks.

The design assumptions for real-time event-based communication typically include access to a network infrastructure, a known upper bound on the number of participating entities, and known resource requirements. Infrastructure networks have an implicit assumption of known connectivity in the absence of failed network components [110]. Real-time event models often assume a known maximum number of entities connected to the physical medium as well as known resource requirements for the communication between these entities [111]. As a result of these assumptions, event transmission schedules for avoiding collisions are typically planned statically and the correctness of these schedules regarding temporal overlaps are verified off-line.

The characteristics of mobile applications using ad hoc wireless networks that render these assumptions inappropriate include dynamic connectivity, unpredictable latency, and limitations to the available resources. The quality of the connections between mobile entities is directly related to entity migration and hence, may vary over time. Entity migration and the fact that mobile applications typically comprise changing populations of entities leads to connections being established highly dynamically. These variations in connection quality in combination with the possibility of colliding transmissions caused by multiple entities simultaneously accessing the shared wireless medium result in unpredictable routing latencies that prevent static transmission planning.

A technique for achieving timeliness and reliability for real-time event-based communication in ad hoc wireless networks has been proposed by Hughes and Cahill [87]. Their conceptual model is the first to directly address the issue of achieving timeliness and reliability in

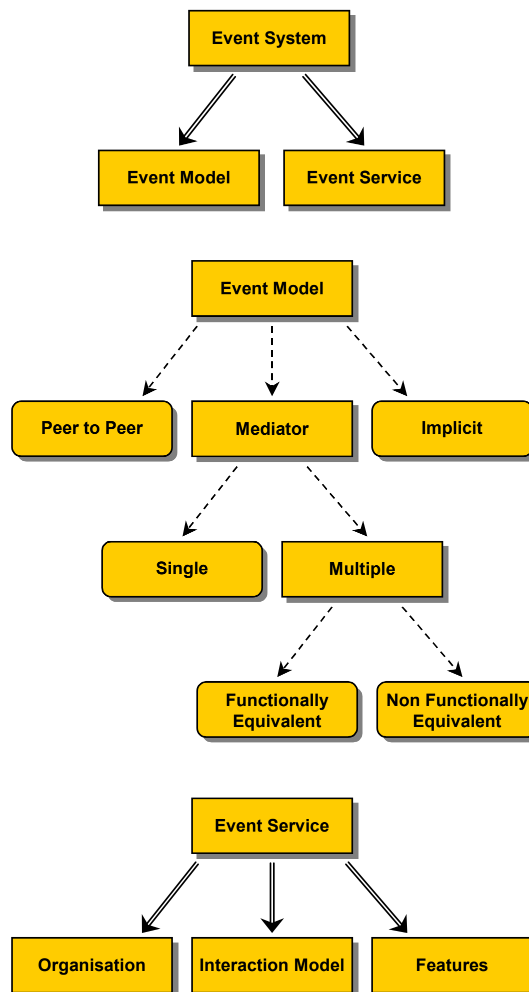
dynamic networks and essentially relies on predictive techniques to alleviate the impediments to real-time event-based communication that are characteristic of mobile ad hoc environments. This model exploits STEAM's concept of bounding the event propagation range for dividing a large, highly dynamic network topology into smaller and therefore less dynamic topologies. This model essentially allows the non-functional requirements of an event type to be mapped to a quality of service zone that is defined by the associated proximity. The focus of this model is then to use predictive techniques to reduce reaction to mobility and topological changes. A proactive technique based on the ability to predict entity and indeed proximity movements is used for reserving the network resources required to achieve probabilistic guarantees on path availability for routing events from a producer to consumers. The techniques proposed by this quality of service model therefore represent a possible approach for enforcing the classes of events that are supported by STEAM.

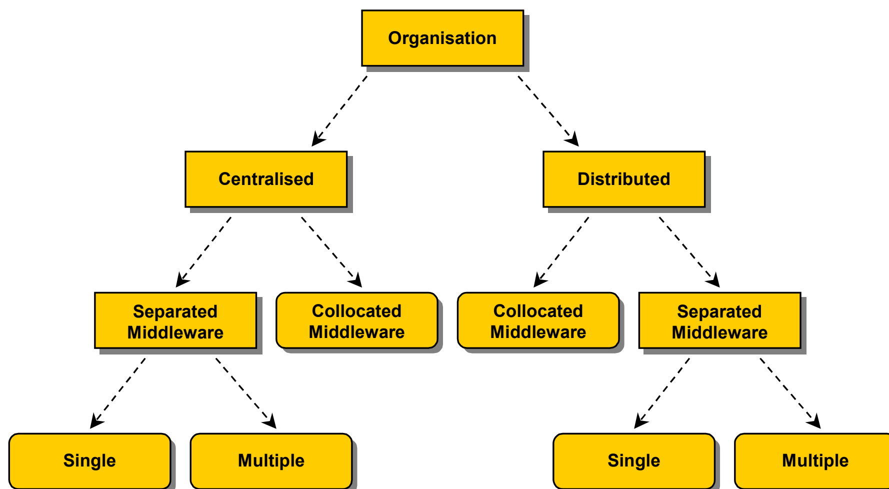
7.3 Conclusion

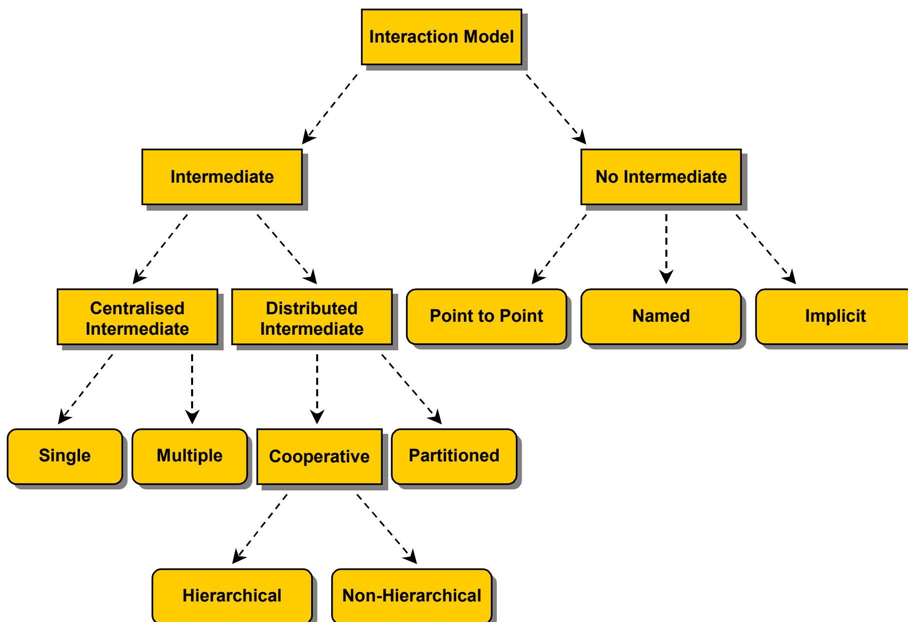
This chapter summarised the motivations for and the most significant achievements of the work presented in this thesis. In particular, it outlined how this work contributes to the state of the art in distributed event-based programming system for the mobile computing domain by providing an event-based middleware for accommodating collaborative applications that use wireless ad hoc networks. The chapter was concluded with some suggestions for possible future work arising from the research undertaken as part of this thesis.

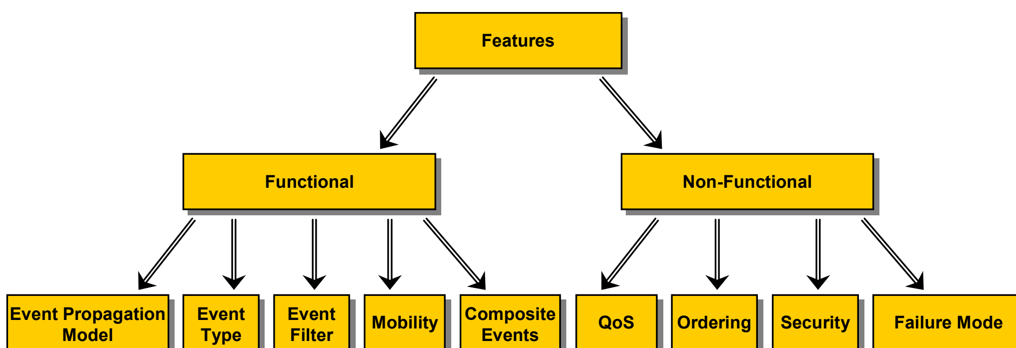
APPENDIX: SUMMARY OF TAXONOMY PROPERTIES

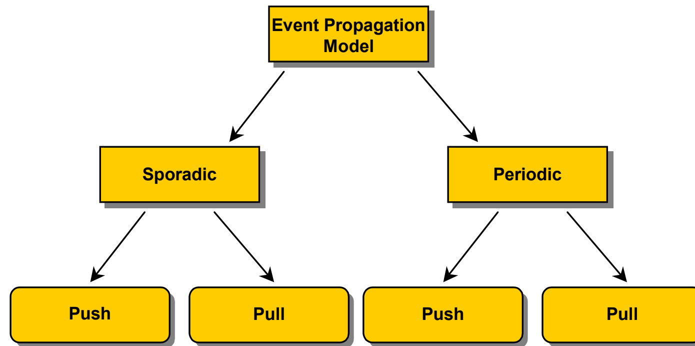
This appendix summarises the taxonomy of distributed event-based programming systems presented in chapter 3 of this thesis. The identified hierarchy of event system properties is presented in this appendix in order to allow for convenient use of the taxonomy when classifying distributed event-based programming systems.

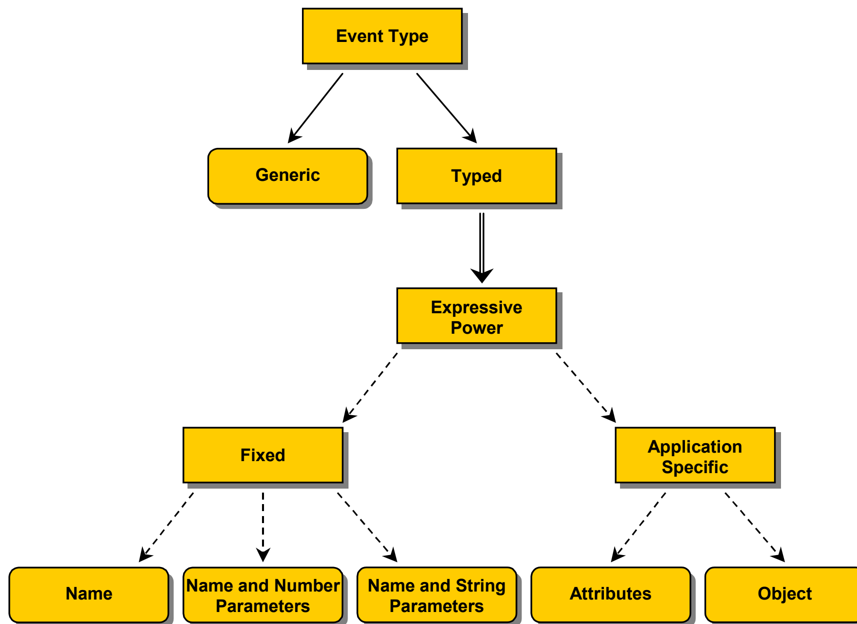


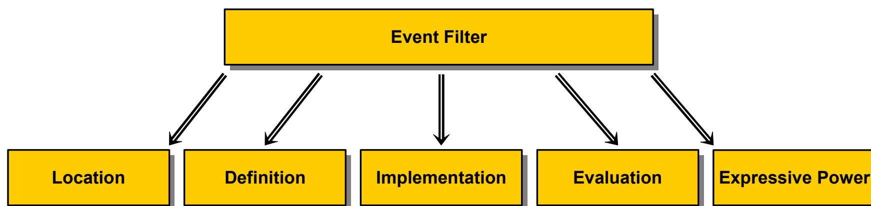

Organisation
Centralised
Distributed
Separated Middleware
Collocated Middleware
Collocated Middleware
Separated Middleware
Single
Multiple
Single
Multiple

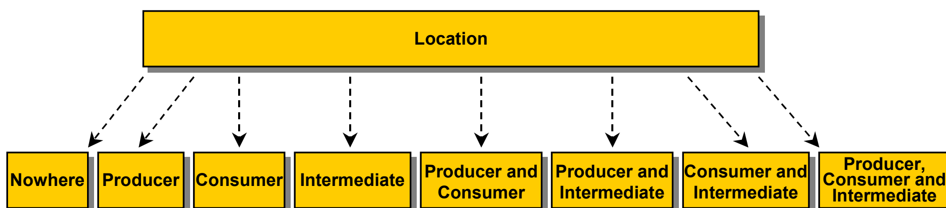

Interaction Model
Intermediate
No Intermediate
Centralised Intermediate
Distributed Intermediate
Point to Point
Named
Implicit
Single
Multiple
Cooperative
Partitioned
Hierarchical
Non-Hierarchical

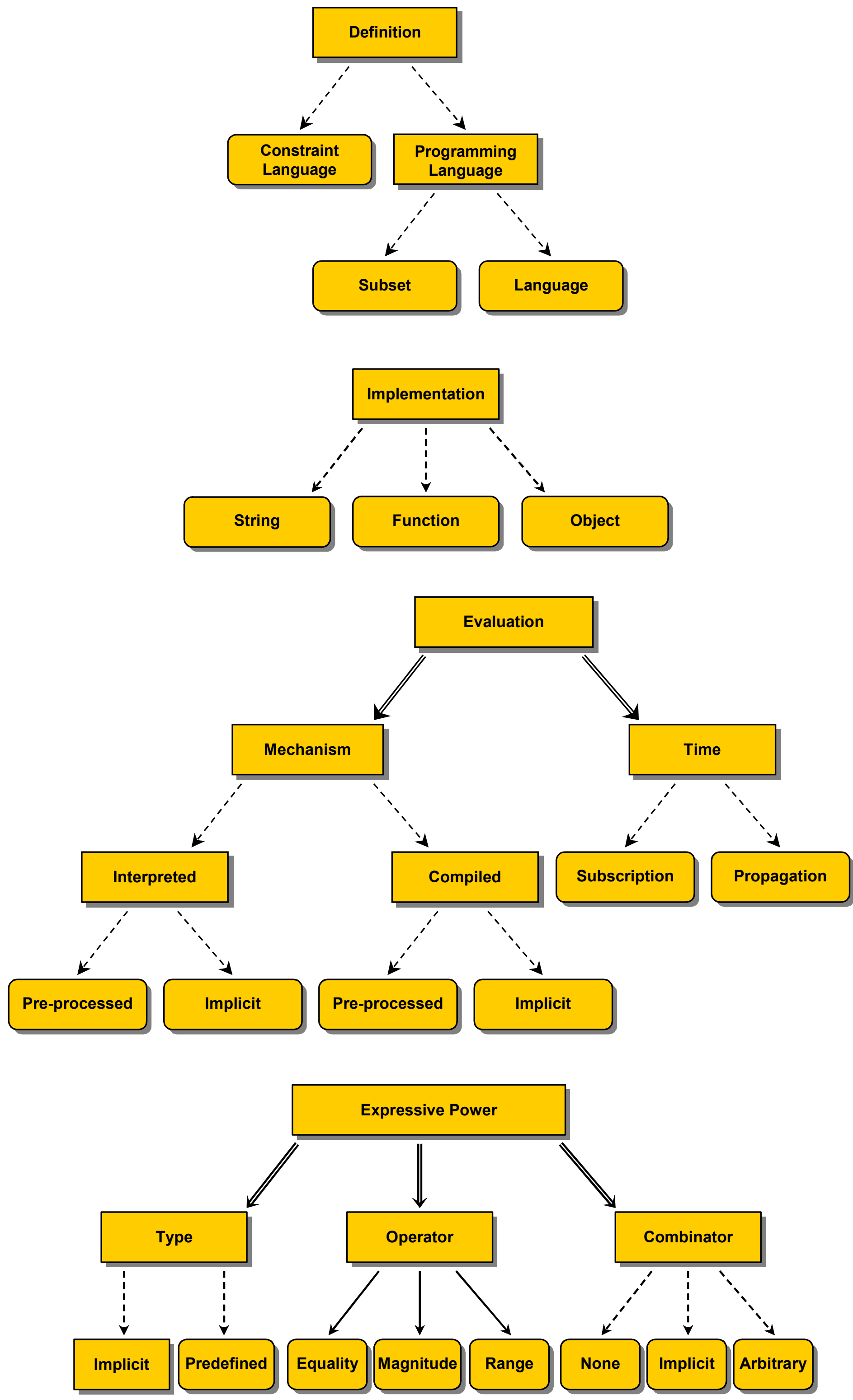

Features
Functional
Non-Functional
Event Propagation Model
Event Type
Event Filter
Mobility
Composite Events
QoS
Ordering
Security
Failure Mode

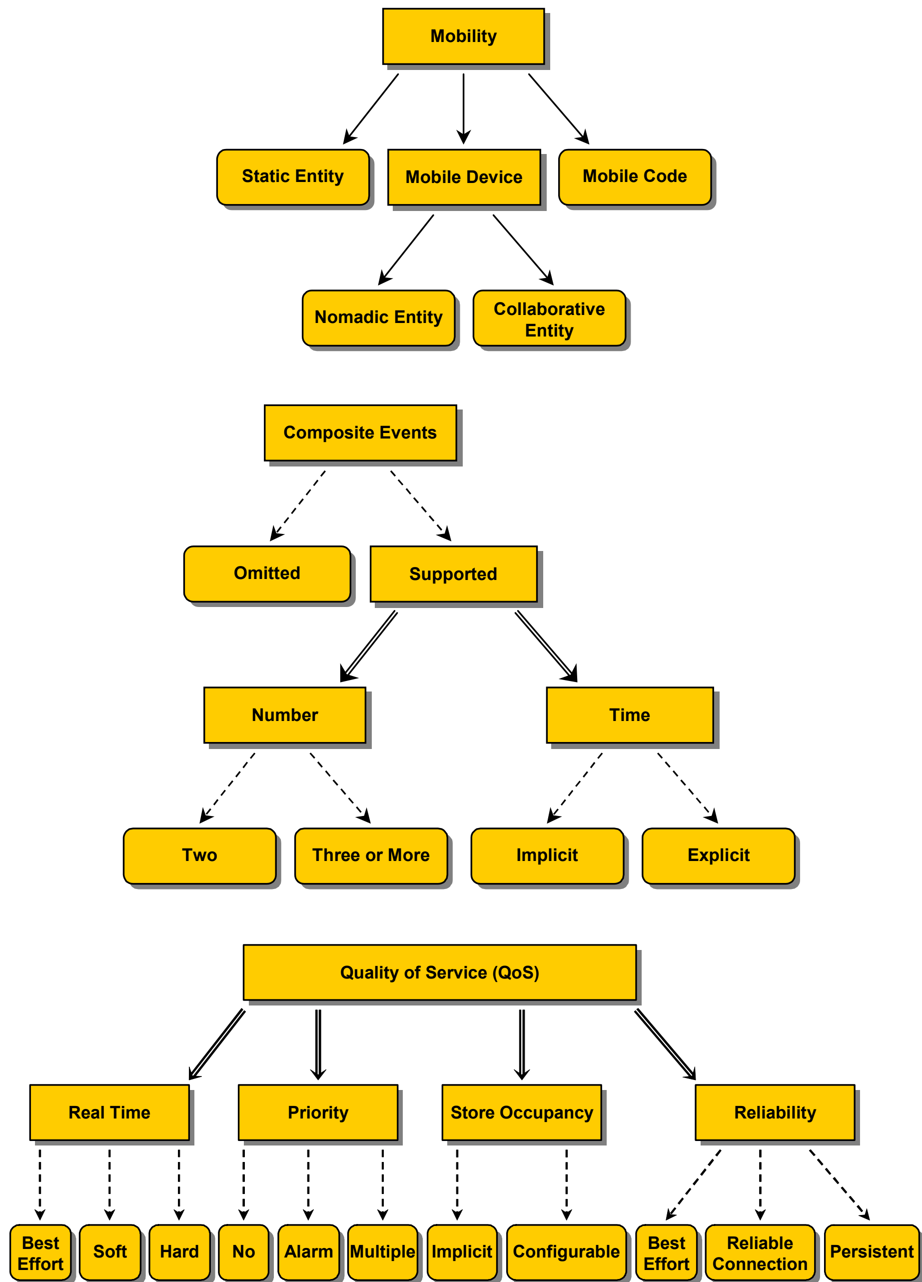

Event Propagation Model
Sporadic
Periodic
Push
Pull
Push
Pull

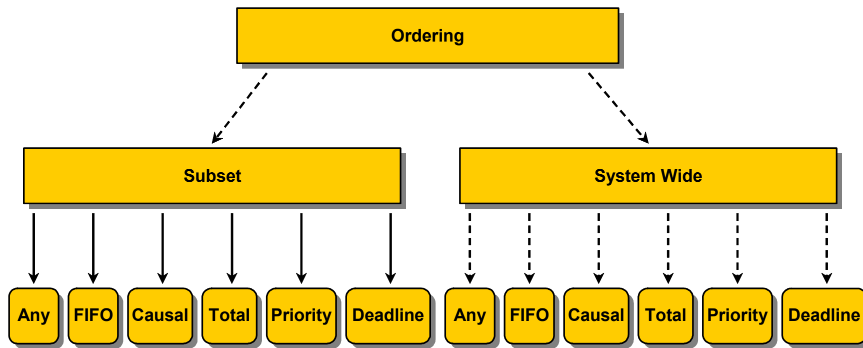

Event Type
Generic
Typed
Expressive Power
Fixed
Application Specific
Name
Name and Number Parameters
Name and String Parameters
Attributes
Object

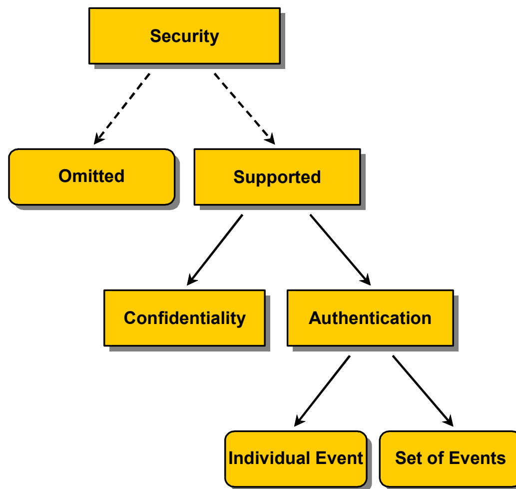

Event Filter
Location
Definition
Implementation
Evaluation
Expressive Power

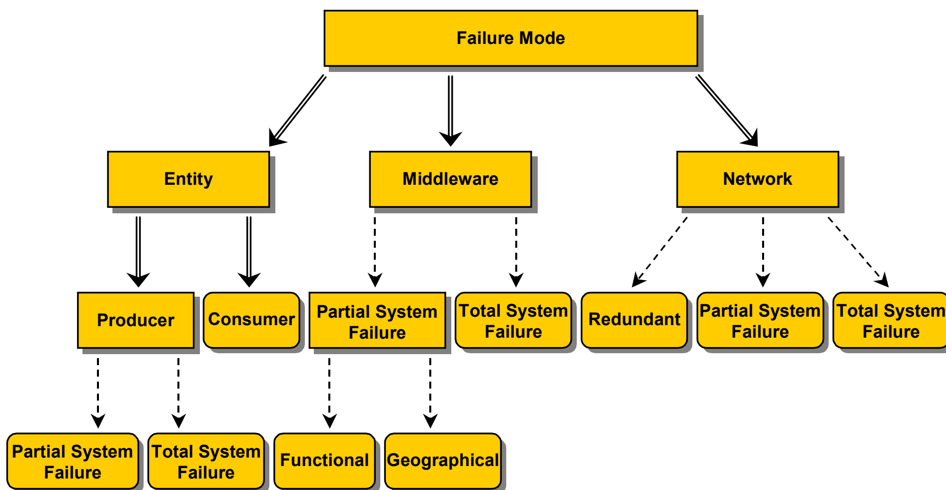

Location
Nowhere
Producer
Consumer
Intermediate
Producer and Consumer
Producer and Intermediate
Consumer and Intermediate
Producer, Consumer and Intermediate


Definition
Constraint Language
Programming Language
Subset
Language
Implementation
String
Function
Object
Evaluation
Mechanism
Time
Interpreted
Compiled
Subscription
Propagation
Pre-processed
Implicit
Pre-processed
Implicit
Expressive Power
Type
Operator
Combinator
Implicit
Predefined
Equality
Magnitude
Range
None
Implicit
Arbitrary


Mobility
Static Entity
Mobile Device
Mobile Code
Nomadic Entity
Collaborative Entity
Composite Events
Omitted
Supported
Number
Time
Two
Three or More
Implicit
Explicit
Quality of Service (QoS)
Real Time
Priority
Store Occupancy
Reliability
Best Effort
Soft
Hard
No
Alarm
Multiple
Implicit
Configurable
Best Effort
Reliable Connection
Persistent


Ordering
Subset
System Wide
Any
FIFO
Causal
Total
Priority
Deadline
Any
FIFO
Causal
Total
Priority
Deadline


Security
Omitted
Supported
Confidentiality
Authentication
Individual Event
Set of Events


Failure Mode
Entity
Middleware
Network
Producer
Consumer
Partial System Failure
Total System Failure
Redundant
Partial System Failure
Total System Failure
Partial System Failure
Total System Failure
Functional
Geographical

BIBLIOGRAPHY

[1] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems, Concepts and Design*, Third ed: Pearson Education Limited, 2001.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*: Object Management Group, 1995.

[3] Sun Microsystems Inc., *Java Remote Method Invocation (RMI) Specification*: Sun Microsystems Inc., 1996.

[4] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri, "Generic Support for Distributed Applications," *IEEE Computer*, vol. 33, pp. 68-76, 2000.

[5] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering (TSE)*, vol. 27, pp. 827-850, 2001.

[6] Y. Huang and H. Garcia-Molina, "Publish/Subscribe in a Mobile Environment," in *Proceedings of the Second ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDe'01)*. Santa Barbara, CA, USA, 2001, pp. 27-34.

[7] R. Meier, "Communication Paradigms for Mobile Computing," *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 6, pp. 56-58, 2002.

[8] H.-A. Jacobsen, "Middleware Services for Selective and Location-based Information Dissemination in Mobile Wireless Networks," presented at Advanced Topic Workshop on Middleware for Mobile Computing (IFIP/ACM Middleware 2001), Heidelberg, Germany, 2001.

[9] B. P. Crow, I. Widjaja, J. G. Kim, and P. T. Sakai, "IEEE 802.11 Wireless Local Area Networks," *IEEE Communications Magazine*, pp. 116-126, 1997.

[10] M. Weiser, "Ubiquitous Computing," *IEEE Hot Topics*, vol. 26, pp. 71-72, 1993.

[11] P. Verissimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday, and J. Kaiser, "CORTEX: Towards Supporting Autonomous and Cooperating Sentient Entities," in *Proceedings of the European Wireless Conference*. Florence, Italy, 2002.

[12] Object Management Group, *CORBAservices: Common Object Services Specification - Notification Service Specification, Version 1.0*: Object Management Group, 2000.

[13] Sun Microsystems Inc., *Java Distributed Event Specification*: Sun Microsystems Inc., 1998.

[14] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul, "Filtering and Scalability in the ECO Distributed Event Model," in *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (ICSE/PDSE 2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 83-95.

[15] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 283 - 331, 2001.

[16] I. Podnar, M. Hauswirth, and M. Jazayeri, "Mobile Push: Delivering Content to Mobile Users," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 563-570.

[17] P. Sutton, R. Arkins, and B. Segall, "Supporting Disconnectedness – Transparent Information Delivery for Mobile and Invisible Computing," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*. Brisbane, Australia: IEEE CS Press, 2001, pp. 277-285.

[18] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 639-644.

[19] R. Meier and V. Cahill, "Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications," in *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, *LNCS 2893*. Paris, France: Springer-Verlag Heidelberg, Germany, 2003, pp. 285-296.

[20] D. Chambers, G. Lyons, and J. Duggan, "Design of Virtual Store using Distributed Object Technology," in *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE/ICSE 2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 66-75.

[21] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper, "Implementing a Sentient Computing System," *IEEE Computer*, vol. 34, pp. 50-56, 2001.

[22] H. Muller and C. Randell, "An Event-Driven Sensor Architecture for Low Power Wearables," in *Proceedings of the Workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC/ICSE2000)*. Limerick, Ireland: IEEE Computer Society, 2000, pp. 39-41.

[23] R. Meier and V. Cahill, "Taxonomy of Distributed Event-Based Programming Systems," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 585-588.

[24] Sun Microsystems Inc., *Java AWT: Delegation Event Model*: Sun Microsystems Inc., 1997.

[25] Microsoft Corporation, *C# Language Specification, Version 0.28*: Microsoft Corporation, 2001.

[26] S. Maffeis, "Developing Publish/Subscribe Applications with iBus," SoftWired AG, White Paper 1999.

[27] M. Erzberger and M. Altherr, "Every Dad Needs a Mom - Message-Oriented Middleware," SoftWired AG, White Paper 1999.

[28] S. J. Kang, S. H. Park, and J. H. Park, "ROOM-BRIDGE: A Vertically Configurable Network Architecture and Real-Time Middleware for Interoperability between Ubiquitous Consumer Devices in Home," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*. Heidelberg, Germany: Springer-Verlag, 2001, pp. 232-251.

[29] R. Meier and V. Cahill, "Location-Aware Event-Based Middleware: A paradigm for Collaborative Mobile Applications?," presented at the 8th CaberNet Radicals Workshop, Ajaccio, Corsica, France, 2003.

[30] Object Management Group, *CORBAservices: Common Object Services Specification - Event Service Specification*: Object Management Group, 1995.

[31] T. Harrison, D. Levine, and D. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," in *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. Atlanta, Georgia, USA: ACM Press, 1997, pp. 184-200.

[32] J. Orvalho, L. Figueiredo, and F. Boavida, "Evaluating Light-weight Reliable Multicast Protocol Extensions to the CORBA Event Service," in *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*. University of Mannheim, Germany, 1999.

[33] J. Bacon, J. Bates, R. Hayton, and K. Moody, "Using Events to Build Distributed Applications," in *Proceedings of the Second International Workshop on Services in Distributed and Networked Environments (SDNE'95)*. Whistler, British Columbia, Canada, 1995, pp. 148-155.

[34] K. O'Connell, T. Dinneen, S. Collins, B. Tangney, N. Harris, and V. Cahill, "Techniques for Handling Scale and Distribution in Virtual Worlds," in *Proceedings of the Seventh ACM SIGOPS European Workshop*. Connemara, Ireland: ACM Press, 1996, pp. 17-24.

[35] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content Based Routing with Elvin4," in *Proceedings of AUUG2K*. Canberra, Australia, 2000.

[36] S. Maffeis, "Client/Server Term Definition," in *Encyclopedia of Computer Science, 4th Edition*, A. Ralston, D. Hemmendinger, and E. Reilly, Eds.: International Thomson Computer Publishing, 2000.

[37] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," in *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*. Austin, TX, USA, 1999, pp. 262-272.

[38] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman, "Exploiting IP Multicast in Content-Based Publish-Subscribe Systems," in *Proceedings of IFIP/ACM International Conference on Distributed Processing (Middleware 2000)*. New York, USA: Springer-Verlag, 2000, pp. 185-207.

[39] I. Sommerville, *Software Engineering*: Addison Wesley, 1995.

[40] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching Events in a Content-based Subscription System," in *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*. Atlanta, GA, USA, 1999, pp. 53-61.

[41] C. Krishna and K. Shin, *Real-Time Systems*: The McGraw-Hill Companies, Inc., 1997.

[42] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*: Addison Wesley Longman Limited, 1996.

[43] *Oxford Dictionary of Computing*, Fourth ed. Oxford: Oxford University Press, 1996.

[44] S. Mullender, *Distributed Systems*: Addison Wesley, 1993.

[45] Object Management Group, *CORBAservices: Notification Service Specification – Request For Proposal*: Object Management Group, 1996.

[46] BEA Systems et al., "CORBAservices: Notification Service Specification - Joint Revised Submission," Object Management Group, Technical Report Telecom 98-11-01, November 1998.

[47] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 3.0; Chapter 3: OMG IDL Syntax and Semantics*: Object Management Group, 2002.

[48] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 3.0; Chapter 13: ORB Interoperability Architecture*: Object Management Group, 2002.

[49] Object Management Group, *CORBAservices: Common Object Services Specification - Trading Object Service Specification*: Object Management Group, 1997.

[50] Object Management Group, *CORBAservices: Management of Event Networks – Request For Proposal*: Object Management Group, 1998.

[51] Object Management Group, *CORBAservices: Common Object Services Specification - Management of Event Domains Specification*: Object Management Group, 2000.

[52] D. C. Schmidt, "Real-Time CORBA with TAO (The ACE ORB)," http://www.cs.wustl.edu/~schmidt/TAO.html, 2003.

[53] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 3.0; Chapter 15: General Inter-ORB Protocol*: Object Management Group, 2002.

[54] D. M. Geary, "Chapter 9, The Delegation Event Model (AWT 1.1 and Beyond)," in *Graphic Java 2, Mastering the JFC: AWT*, vol. 1, 2001.

[55] Sun Microsystems Inc., *JavaBeans Specification*: Sun Microsystems Inc., 1997.

[56] Sun Microsystems Inc., *Swing Component Set*: Sun Microsystems Inc., 1998.

[57] Sun Microsystems Inc., *EmbeddedJava Specification*: Sun Microsystems Inc., 1999.

[58] Sun Microsystems Inc., *PersonalJava Specification*: Sun Microsystems Inc., 2000.

[59] Sun Microsystems Inc., *Jini: Distributed Event Specification*: Sun Microsystems Inc., 1999.

[60] Sun Microsystems Inc., *Java Distributed Leasing Specification*: Sun Microsystems Inc., 1998.

[61] Microsoft Corporation, *Distributed Component Object Model (DCOM) Architecture*: Microsoft Corporation, 1997.

[62] C. Ma and J. Bacon, "COBEA: A CORBA-Based Event Architecture," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. Santa Fe, New Mexico, USA, 1998, pp. 117-131.

[63] P. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," in *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*. Vienna, Austria, 2002, pp. 611-618.

[64] P. Pietzuch, B. Shand, and J. Bacon, "A Framework for Event Composition in Distributed Systems," in *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Middleware (Middleware 2003)*. Rio de Janeiro, Brazil, 2003.

[65] V. Cahill, A. Condon, D. Kelly, S. McGerty, K. O'Connell, G. Starovic, and B. Tangney, "MOONLIGHT: VOID Shell Specification," Dept. of Computer Science, Trinity College Dublin, Ireland, Technical Report TCD-CS-95-15, 1995.

[66] K. O'Connell, V. Cahill, A. Condon, S. McGerty, G. Starovic, and B. Tangney, "The VOID Shell: A Toolkit for The Development of Distributed Video Games and Virtual Worlds," in *Proceedings of the Workshop on Simulation and Interaction in Virtual Environments*. University of Iowa, Iowa City, USA, 1995, pp. 172-177.

[67] M. Haahr, "Implementation and Evaluation of Scalability Techniques in the ECO Model," in *Dept. of Computer Science*: Trinity College Dublin, Ireland, 1998.

[68] G. Starovic, V. Cahill, and B. Tangney, "An Event Based Object Model for Distributed Programming," in *Proceedings of the International Conference on Object Oriented Information System*. London, UK: Springer-Verlag, 1995, pp. 72-86.

[69] K. Birman, *Building Secure and Reliable Network Applications*: Manning Publishing Co., 1996.

[70] K. O'Connell, "System Support for Distributed Multi-User Virtual Worlds," in *Dept. of Computer Science*: Trinity College Dublin, Ireland, 1997.

[71] G. Cugola and C. Ghezzi, "The Design and Implementation of PROSYT: An Experience in Developing an Event-Based, Mobile Application," in *Proceeding of the IEEE 8th International Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises (IEEE WET ICE'99)*. Stanford University, Stanford, California, USA, 1999.

[72] M. Caporuscio, A. Carzaniga, and A. L. Wolf, "Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications," Department of Computer Science, University of Colorado, Boulder, Colorado, USA, Technical Report CU-CS-944-03, January 2003.

[73] M. Caporuscio, A. Carzaniga, and A. L. Wolf, "An Experience in Evaluating Publish/Subscribe Services in a Wireless Network," in *Proceeding of the Third International Workshop on Software and Performance (ISSTA/WOSP 2002)*. Rome, Italy, 2002.

[74] B. Segall and D. Arnold, "Elvin Has Left The Building: A Publish/Subscribe Notification Service With Quenching," in *Proceedings of AUUG97*. Brisbane, Australia, 1997.

[75] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann, "Engineering Event-Based Systems with Scopes," in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*. Málaga, Spain: Springer-Verlag, 2002, pp. 309-333.

[76] B. Martin, C. Pedersen, and J. Bedford-Roberts, "An Object-Based Taxonomy for Distributed Computing Systems," *IEEE Computer*, vol. 24, pp. 17-27, 1991.

[77] D. J. Barrett, L. A. Clarke, P. L. Tarr, and A. E. Wise, "A Framework for Event-based Software Integration," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, pp. 378 - 421, 1996.

[78] D. S. Rosenblum and A. L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," in *Proceedings of the The Fifth Symposium on the Foundations of Software Engineering (FSE5) and The Sixth European Software Engineering Conference (ACM SIGSOFT ESEC97)*. Zurich, Switzerland, 1997, pp. 344-360.

[79] Iona Technologies, "Orbix 3 Product Family," Iona Technologies, White Paper April 1999.

[80] A. Padovitz, S. W. Loke, and A. B. Zaslavsky, "Using the Publish-Subscribe Communication Genre for Mobile Agents," in *Proceedings of the First German Conference on Multiagent System Technologies (MATES'03)*, *LNCS 2831*. Erfurt, Germany: Springer-Verlag Heidelberg, Germany, 2003, pp. 180-191.

[81] S. W. Loke, A. Padovitz, and A. B. Zaslavsky:, "Context-Based Addressing: The Concept and an Implementation for Large-Scale Mobile Agent Systems," in *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, *LNCS 2893*. Paris, France: Springer-Verlag Heidelberg, Germany, 2003, pp. 274-284.

[82] J. Bacon, J. Bates, R. Hayton, and K. Moody, "Using Events to Build Distributed Applications," in *Proceedings of the Seventh ACM SIGOPS European Workshop*. Connemara, Ireland, 1996, pp. 9-16.

[83] C. Liebig, M. Cilia, and A. Buchmann, "Event Composition in Time-Dependent Distributed Systems," in *Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*. Edinburgh, Scotland, 1999, pp. 70-78.

[84] J. Kaiser, C. Brudna, C. Mitidieri, and C. Pereira, "COSMIC: A Middleware for Event-Based Interaction on CAN," in *Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2003)*. Lisbon, Portugal, 2003.

[85] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf, "Security Issues and Requirements for Internet-scale Publish-Subscribe Systems," in *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS)*. Big Island, Hawaii, USA, 2002.

[86] N. Reijers, R. Cunningham, R. Meier, B. Hughes, G. Gaertner, and V. Cahill, "Using Group Communication to Support Mobile Augmented Reality Applications," in *Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*. Crystal City, VA, USA, 2002, pp. 297-306.

[87] B. Hughes and V. Cahill, "Towards Real-time Event-based Communication in Mobile Ad Hoc Wireless Networks," in *Proceedings of 2nd International Workshop on Real-Time LANS in the Internet Age 2003 (ECRTS/RTLIA03)*. Porto, Portugal, 2003, pp. 77-80.

[88] F. Cristian, "Synchronous and Asynchronous Group Communication," *Communications of the ACM*, vol. 39, pp. 88-97, 1996.

[89] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," *ACM Computing Surveys*, vol. 33, pp. 427-496, 2001.

[90] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A Case for Message Oriented Middleware," presented at Proceedings of the 13th International Symposium on DIStributed Computing (DISC'99), Bratislava, Slovak Republic, 1999.

[91] M. O. Killijian, R. Cunningham, R. Meier, L. Mazare, and V. Cahill, "Towards Group Communication for Mobile Participants," in *Proceedings of Principles of Mobile Computing (POMC'2001)*. Newport, Rhode Island, USA, 2001, pp. 75-82.

[92] R. Meier, M. O. Killijian, R. Cunningham, and V. Cahill, "Towards Proximity Group Communication," presented at Advanced Topic Workshop on Middleware for Mobile Computing (IFIP/ACM Middleware 2001), Heidelberg, Germany, 2001.

[93] Y.-B. Ko and N. H. Vaidya, "GeoTORA: A Protocol for Geocasting in Mobile Ad Hoc Networks," in *Proceedings of the 8th International Conference on Network Protocols (ICNP 2000)*. Osaka, Japan, 2000.

[94] G.-C. Roman, Q. Huang, and A. Hazemi, "Consistent Group Membership in Ad Hoc Networks," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2002)*. Toronto, Canada, 2001, pp. 381-388.

[95] R. Cunningham and V. Cahill, "Time Bounded Medium Access Control for Ad Hoc Networks," in *Proceedings of the Second ACM International Workshop on Principles of Mobile Computing (POMC'02)*. Toulouse, France: ACM Press, 2002, pp. 1-8.

[96] T. Goff, N. B. Abu-Ghazaleh, D. S. Phatak, and R. Kahvecioglu, "Preemptive Routing in Ad Hoc Networks," in *Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM 2001)*. Rome, Italy, 2001, pp. 43-52.

[97] K. Paul, S. Bandyopadhyay, A. Mukherjee, and D. Saha, "Communication-Aware Mobile Hosts in Ad-Hoc Wireless Network," in *Proceedings of the International Conference on Personal Wireless Communications (ICPWC'99)*. Jaipur, India, 1999, pp. 83-87.

[98] L. Qin and T. Kunz, "Pro-Active Route Maintenance in DSR," *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 6, pp. 79-89, 2002.

[99] G. Gaertner, E. O'Nuallain, A. Butterly, K. Singh, and V. Cahill, "802.11 Link Quality and its Prediction - An Experimental Study," 2004, submitted for publication.

[100] B. O'Hara and A. Petrick, *The IEEE 802.11 Handbook: A Designer's Companion*: Standards Information Network IEEE Press, 1999.

[101] B. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*: John Wiley & Sons, Inc., 1999.

[102] P. S. Wang, *C++ with Object-Oriented Programming*: PWS Publishing Company, 1994.

[103] S.-J. Lee, W. Su, J. Hsu, M. Gerla, and R. Bagrodia, "A Performance Comparison Study of Ad Hoc Wireless Multicast Protocols," in *Proceedings of IEEE INFOCOM 2000*. Tel Aviv, Israel, 2000.

[104] C. E. Perkins, E. M. Royer, S. R. Das, and M. K. Marina, "Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks," *IEEE Personal Communications Magazine Special Issue on Mobile Ad Hoc Networks*, vol. 8, pp. 16-29, 2001.

[105] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva, "A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols," in *Proceedings of the 4th Annual International Conference on Mobile Computing and Networking*. Dallas, Texas, USA, 1998.

[106] Z. J. Haas, J. Deng, B. Liang, P. Papadimitratos, and S. Sajama, "Wireless Ad Hoc Networks," in *Wiley Encyclopedia of Telecommunications*, J. G. Proakis, Ed. New York: John Wiley & Sons, 2002.

[107] Z. Haas, J. Y. Halpern, and L. Li, "Gossip-Based Ad Hoc Routing," in *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2002)*. New York City, USA, 2002, pp. 1707-1716.

[108] N. Priyantha, A. Chakraborty, and H. Balakrishnan, "The Cricket Location-Support System," in *Proceedings of the 6th ACM/IEEE Annual International Conference on Mobile Computing and Networking (ACM MOBICOM 2000)*. Boston, Massachusetts, USA: ACM Press, 2000, pp. 32-43.

[109] Dublin City Council, "http://www.dublincity.ie," Dublin City Council, Civic Offices, Wood Quay, Dublin 8, Ireland, 2003.

[110] R. Ramanathan and M. Steenstrup, "Hierarchically-Organized, Multihop Mobile Wireless Networks for Quality-of-Service Support," *IEEE Mobile Networks and Applications*, vol. 3, pp. 101-119, 1998.

[111] J. Kaiser and M. Mock, "Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)," in *Proceedings of the 2nd International Symposium on Object-oriented Real-time distributed Computing (ISORC99)*. Saint-Malo, France, 1999, pp. 172-181.