Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware

by

Pablo Ezequiel Guerrero

Submitted to the

Departamento de Computación y Sistemas

Facultad de Ciencias Exactas

in partial fulfillment of the requirement for the degree of

Systems Engineer

(Major in Computer Science)

at the

Universidad Nacional del Centro de la Provincia de Buenos Aires

Thesis Supervisor:	Mariano Cilia
Date:	September, 2004

2004, Tandil, República Argentina

Acknowledgements

First of all I would like to thank my advisor Prof. Dr. Mariano Cilia for his invaluable help and advice. His guidance during my stay at Darmstadt, Germany, was of great importance to me. He did not only introduce me to the distributed systems world, but enormously helped me to prepare for a career in research.

Second, I want to thank and express my great appreciation for the support and funding from the Databases and Distributed Systems Group, as well as from the Technische Universität Darmstadt (TUD). This funding allowed me to visit Germany, a country where technology occupies an important place and is fortunately applied, which enabled me to start my thesis work very motivated.

I want to thank the people at the DVS Group, particularly to Alejandro Buchmann, Patric Kabus, Samuel Kounev, Ludger Fiege, Andreas Zeidler, Christian Haul, Matthias Meixner and Marion Braun. They supported me in different ways from getting to know the TU-D to develop very particular aspects of my work.

Special thanks go to Mario and José Antollini. They have not only been the best colleagues in the most important parts of the universitary career at the UNICEN, but also big pals in my daily student life at Tandil during the last years.

I also would like to thank to Jane Pryor and Claudia Marcos from the ISISTAN Institute for the time spent with discussions about Aspect Oriented Software Development.

The best thanks go to my family. My parents José Luis and Martha were, are and will always be my best advisors and teachers. My father's perseverance encouraged me to aim high and work hard, whereas my mother's continuous love and support allowed me to concentrate in the career without forgetting what really matters. Obviously I also want to thank my sister Mariel for her just-so-cool support.

Last, but not least, I want to thank my woman Luciana. Her unconditional love has always inspired me to be a better individual. She supported me since the beginning of our relationship by traveling thousands of times to get together, and her awesome support during my stay at Germany is simply something I'll never forget.

Abstract

Nowadays we are experiencing a convergence of technologies that results in an explosion of information. This phenomenon requires new paradigms for adequate data and information management and processing. The World Wide Web is a huge source of information that was conceived for interactive search by humans. To exploit the Web in a mode other than human browsing confronts us with the need for filtering and interpreting a large amount of heterogeneous and often short-lived data. The deployment of smart devices requires the continuous monitoring of events as well as the appropriate context information to interpret them properly. The miniaturization of sensors and their ubiquitous deployment will result in massive amounts of sensor signals that must be processed, often in real time. Huge distributed systems must be capable of detecting and correcting failures and return autonomously to stable operation. New business strategies, such as *event-driven supply chain management* and *zero-latency enterprises*, depend on the timely dissemination of information and business events.

Common to the above is that signals and data, which we abstract into the notion of *events* and *event notifications*, will flow from producers to consumers. When dealing with event streams our traditional, pull-based, *request/reply* access mechanisms to stagnant data do no longer work [16].

There is a fundamental difference between traditional applications and the scenarios described when analyzing the information space and its interactions [14]. In the well-known request-reply interaction pattern *(pull-based)*, the interaction is initiated by the client or consumer of information and the request is directed at a specific server or information provider (who provides the information). On the other hand, when the information flows, the interaction is started *by the producer* of the information *(push-based)*. If the producer knows the identity of its counterpart, we have a classical messaging interaction. However, if the producer does not know the consumer a priori, we have the typical event-based interaction that depends on a mediator or broker to connect the interested parties, frequently called *middleware*.

These new interaction patterns found on previously described emergent applications require an adequate treatment through appropriate mechanisms. These must be based on an infrastructure that is able to support a growing information flow, generated by a huge quantity of interconnected devices, services and applications with different capabilities that will react and automate processes on our behalf. Streaming events must be detected, interpreted, aggregated, filtered, analyzed, and reacted to interesting situations based on the information flow.

Together, the emergence of *mobile computing* has opened a whole new set of services for the benefit of the mobile user. A convenient way to build these systems is using event-based infrastructures, particularly with *publish/subscribe* middleware. These systems provide asynchronous communications, naturally decouple producers and consumers, make them anonymous to each other and allows a dynamic number of publishers and subscribers. It also provides support for *roaming clients*, e.g., to bridge phases of disconnection, and a notion of *location* tailored for efficient location-dependent information delivery.

One of the problems that many publish/subscribe applications must deal with is found in their runtime startup. These applications have to be bootstrapped to correctly interpret the current flow of notifications and commence normal operation. This problem is aggravated in mobile environments where disconnections and context changes occur frequently, basically to adapt the application to current contextual information which is only available locally. This initial stage is what we call

the *bootstrapping sequence*, and the time it takes is called *bootstrapping latency.* Before this stage is finished an application might not be able to work properly. This work concentrates in enhancing the mobile pub/sub middleware by reducing the bootstrapping latency.

This thesis comprises a theoretical and a practical part. The theoretical part starts by introducing the reader into distributed systems, information-driven applications, mobile computing and publish/subscribe systems. Then we return to the stated problem by describing a generic approach to confront it. Given distinct requirements that the generic solution must overcome, we proceed by further analyzing several strategies that solve the problem in different ways. The practical part focuses first in the design and implementation issues of the analyzed strategies; and then moves into a practical evaluation of the implementation based upon the prototype.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Problem Statement . . . 2

1.3 Proposed Approach . . . 3

1.4 Goals and Contributions of this Thesis . . . 4

1.5 Issues not Addressed in this Thesis . . . 4

1.6 Structure of the Thesis . . . 4

2 Background **5**

2.1 Distributed Systems . . . 5

2.2 Middleware for Mobile Computing . . . 6

2.3 Information-driven Applications . . . 7

2.4 Publish/Subscribe Systems . . . 8

2.4.1 System Structure . . . 9

2.4.2 Addressing Models . . . 10

2.4.3 Publish/Subscribe Systems with Advertisements . . . 12

2.5 The Rebeca Notification Service . . . 12

2.5.1 Overview . . . 13

2.5.2 Routing Algorithms . . . 15

2.6 Summary . . . 16

3 Proposed Approach **17**

3.1 Scenario Classification . . . 17

3.2 Proposed Approach Analysis . . . 18

3.3 Summary . . . 20

4 Caching Strategies: Analysis **21**

4.1 Caching Strategies Requirements . . . 21

4.2 Caching in Event-Based Systems . . . 22

4.3 Notifications Storage . . . 23

4.4 Cache Querying . . . 27

4.4.1 Local Broker Caching . . . 28

4.4.2 Border Broker Caching . . . 31

4.4.3 Merging Caching . . . 35

4.4.4 Caching with more complex Routing Algorithms . . . 38

4.4.5 Caching with Advertisements . . . 40

4.4.6 Considering the Time Dimension . . . 42

4.5 Summary . . . 42

5 Caching Strategies: Design and Implementation 45
5.1 Architectural approach . . . 45
5.1.1 The OOP way . . . 45
5.1.2 The AOP way . . . 47
5.2 Design and Implementation with AOP . . . 48
5.2.1 Caching Design . . . 49
5.2.2 Generic Behavior: Notification Storage . . . 58
5.2.3 Specialization of the Framework . . . 59
5.3 Caching Strategies: Implementation Details . . . 62
5.3.1 Packaging with Aspects . . . 62
5.3.2 Plug-and-Play Caching . . . 63
5.4 Summary . . . 64

6 Experimental Results 67
6.1 General Setup . . . 67
6.1.1 Broker Topology . . . 67
6.1.2 Characteristics of the consumers . . . 68
6.1.3 Characteristics of the producers . . . 69
6.1.4 Characteristics of the subscriptions and events . . . 69
6.2 Caching Efficacy . . . 69
6.2.1 Local Broker Caching . . . 70
6.2.2 Border Broker Caching . . . 70
6.2.3 Merging Caching . . . 72
6.2.4 Strategies Comparison . . . 72
6.3 Summary . . . 75

7 Conclusions and Future Work 77
7.1 Conclusions . . . 77
7.2 Future Work . . . 78

A Aspect Oriented Software Development 81
A.1 Aspect-Oriented Programming . . . 81
A.2 AspectJ . . . 81
A.2.1 Joinpoints and Pointcuts . . . 82
A.3 AOP and UML . . . 83

List of Figures

1.1 Traditional approach for application bootstrapping . . . 2
1.2 Adaptation of the bootstrapping phase. . . . 3

2.1 Example of a distributed system . . . 5
2.2 The ISO/OSI Reference Model . . . 6
2.3 Publish/Subscribe systems architecture . . . 9
2.4 Large Scale Publish/Subscribe systems architecture . . . 10
2.5 Channel-based Addressing . . . 11
2.6 Subject-based Addressing . . . 11
2.7 Rebeca architecture details . . . 13
2.8 UML use case diagram . . . 15

3.1 Scenario classification . . . 18
3.2 Extending the pub/sub interface . . . 19
3.3 New subscribtion behavior . . . 20

4.1 Buffering structures . . . 24
4.2 Buffering structure . . . 26
4.3 UAG components . . . 28
4.4 Flooding behavior: initial deployment . . . 29
4.5 Flooding behavior: subscription update . . . 29
4.6 Flooding behavior: notification forwarding . . . 30
4.7 Flooding behavior with Local Broker Caching . . . 30
4.8 Simple Routing behavior: initial deployment . . . 31
4.9 Simple Routing behavior: subscription update . . . 32
4.10 Simple Routing behavior: notification forwarding . . . 32
4.11 Simple Routing behavior with Border Broker Caching . . . 32
4.12 Border Broker Cache Querying . . . 33
4.13 Multiple producers scenario . . . 35
4.14 Merging Caching deployment . . . 36
4.15 Covering-based Routing: initial deployment . . . 38
4.16 Covering-based Routing: subscription forwarding (1) . . . 38
4.17 Covering-based Routing: subscription forwarding (2) . . . 39
4.18 Simple Routing with Advertisements: initial deployment . . . 40
4.19 Simple Routing with Advertisements: advertisement forwarding . . . 41
4.20 Simple Routing with Advertisements: subscription forwarding . . . 41
4.21 Notification Service possible configurations . . . 43

5.1 REBECA's routing class hierarchy 46
5.2 The *Decorator* design pattern approach 47
5.3 UML Use Case diagram involving the *Caching* aspect 49
5.4 High level architecture 50
5.5 Pointcut to create `CachingStrategy` objects 51
5.6 Pointcuts to pass the `RoutingTable`s to `CachingStrategy` objects 52
5.7 Pointcut to pass the neighbor `EventProcessor`'s collection to `CachingStrategy` objects 52
5.8 Pointcut to intercept *subscriptions* 53
5.9 Pointcut to intercept *unsubscriptions* 53
5.10 Pointcut to intercept *event processing* 54
5.11 Pointcut for *subscriptions* and *unsubscriptions* forwarding 54
5.12 Pointcut for *event destination* selection 55
5.13 Pointcut to *initialize* a `RoutingEntry` object 56
5.14 Pointcut to *initialize* many `RoutingEntry` objects 57
5.15 The `Caching` aspect 58
5.16 The `BoundedSubscription` aspect 58
5.17 The `HeldRoutingEntry` aspect 59
5.18 The `EventCache` and `EventQueue` classes 60
5.19 The `CachingStrategy` class hierarchy 61
5.20 UML package diagram extended for aspects 63
5.21 Aspect visualizer weaving points 64

6.1 Broker topology with 4 levels 68
6.2 Local Broker Caching Evaluation 71
6.3 Border Broker Caching Evaluation 71
6.4 Merging Caching Evaluation 72
6.5 Caching Strategies Comparative Evaluation (1) 73
6.6 Caching Strategies Comparative Evaluation (2) 74

7.1 Schematic view of the enhanced notification service 78

List of Tables

5.1 Specification of the crosscutting concern *Caching* 48

6.1 Fixed an varied parameters of the setup 67

List of Pseudocodes

4.1 Buffering Structure Data Definition 25
4.2 Processing of a message m 26
4.3 Processing of a subscription's filter F 27
4.4 Processing of an unsubscription's filter F 27

Chapter 1

Introduction

1.1 Motivation

Nowadays we are experiencing a convergence of technologies that results in an explosion of information. This phenomenon requires new paradigms for adequate data and information management and processing. The World Wide Web is a huge source of information that was conceived for interactive search by humans. To exploit the Web in a mode other than human browsing confronts us with the need for filtering and interpreting a large amount of heterogeneous and often short-lived data. The deployment of smart devices requires the continuous monitoring of events as well as the appropriate context information to interpret them properly. The miniaturization of sensors and their ubiquitous deployment will result in massive amounts of sensor signals that must be processed, often in real time. Huge distributed systems must be capable of detecting and correcting failures and return autonomously to stable operation. New business strategies, such as *event-driven supply chain management* and *zero-latency enterprises*, depend on the timely dissemination of information and business events.

Common to the above is that signals and data, which we abstract into the notion of *events* and *event notifications*, will flow from producers to consumers. When dealing with event streams our traditional, pull-based, *request/reply* access mechanisms to stagnant data do no longer work [16].

There is a fundamental difference between traditional applications and the scenarios described when analyzing the information space and its interactions [14]. In the well-known request-reply interaction pattern *(pull-based)*, the interaction is initiated by the client or consumer of information and the request is directed at a specific server or information provider (who provides the information). On the other hand, when the information flows, the interaction is started *by the producer* of the information *(push-based)*. If the producer knows the identity of its counterpart, we have a classical messaging interaction. However, if the producer does not know the consumer a priori, we have the typical event-based interaction that depends on a mediator or broker to connect the interested parties, frequently called *middleware*.

These new interaction patterns found on previously described emergent applications require an adequate treatment through appropriate mechanisms. These must be based on an infrastructure that is able to support a growing information flow, generated by a huge quantity of interconnected devices, services and applications with different capabilities that will react and automate processes on our behalf. Streaming events must be detected, interpreted, aggregated, filtered, analyzed, and reacted to interesting situations based on the information flow.

Together, the emergence of *mobile computing* has opened a whole new set of services for the benefit of the mobile user. A convenient way to build these systems is using event-based infrastructures, particularly with *publish/subscribe* middleware. These systems provide asynchronous communications, naturally decouple producers and consumers, make them anonymous to each other and allows a dynamic number of publishers and subscribers. It also provides support for *roaming clients*, e.g., to bridge phases of disconnection, and a notion of *location* tailored for efficient location-dependent information delivery.

1.2 Problem Statement

Developers must face many problems when designing distributed event-based applications. One of these problems is found at the application's runtime startup. Basically, these applications require an initial phase to correctly interpret the current flow of notifications and commence normal operation. This phase, also known as *bootstrapping phase*, is necessary in order to bring the application into a consistent state. The time involved in performing this phase is called *bootstrapping latency* and it is assumed that before this task is finished an event-based application might not be able to work properly. There are two possibilities to perform a bootstrapping: a) get the current state from a server (if there is such a server and if the applications knows it); or b) consume/observe notifications until the bootstrapping phase is completed. The first case combines both paradigms (request/reply and pub/sub) while the second is a pure pub/sub approach.

In Figure 1.1 we illustrate the problem. In this stage the application subscribes to the user's interests. But since the application needs some notifications before correctly interpreting the current flow of notifications (shown with the arrival of the *meaningful event* in the figure), the application could remain waiting or even blocked.

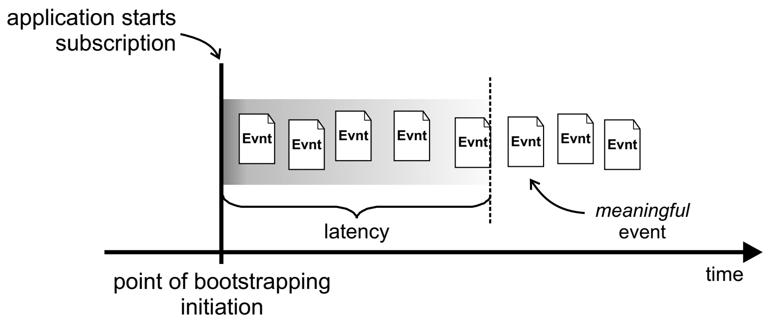

application starts subscription
Evnt
Evnt
Evnt
Evnt
Evnt
Evnt
Evnt
Evnt
latency
meaningful event
point of bootstrapping initiation
time

Figure 1.1: Traditional approach for application bootstrapping

For instance, lets imagine a notification service where several enterprise applications are used to manage exchange values for different currencies. Scenarios like this are very realistic in our country these days. On one hand, organizations like banks, financial services corporations, foreign currency exchange offices, etc., use this service to publish their exchange rate values. Each organization, in turn, might have several brokers authorized to publish new values according to economical trends or other parameters. On the other hand, companies which regularly perform their currency exchange at these organizations benefit with the usage of this service. For example, in order to not loose money, a given wholesaler enforces their salesmen to use the information provided by these organizations when they emit goods orders. Each salesman runs an application in its laptop, PDA or even mobile phone, which they use to subscribe to whatever currencies they are interested in (according to their assigned clients). As it can be seen, only the last published value of the exchange rate is needed in this scenario. But, until the new exchange rate value is published, the application will display a null or useless value. Furthermore, the salesman is constantly visiting different clients, implicitly re-initializing the subscription process (in order to update the notification service about its new position), hence incurring in this problem for each location.

An alternative solution to this problem could be enforcing each organization to publish the last value at fixed intervals, even if the value didn't change. This could effectively bound the waiting time for an initial response. Though, this approach is not always appropriate. Lets imagine another situation within this example where an application wants to build a chart that is a function of the last 10 values of a certain currency exchange rate. Till this amount of values are not available, the application may not work correctly or may show undesired results. In comparison with the other scenario, periodically publishing the last 10 values is not appropriate because it changes the semantics of the published information *order*. Another application would get confused if it is interested in displaying the last 20 values.

As it was shown with the former example, this problem is aggravated in mobile environments. Disconnections and context changes occur frequently, basically to *adapt* the application to current contextual information which is only available locally (we will describe this situations further in section 2.2). In pervasive scenarios, where applications rely on location-dependent information, the fact is that whenever a client reaches a new location (context switch), a new bootstrapping phase must be initiated. In such settings, location-dependent information is required in order to bootstrap location-aware applications. This situation is not an isolated occasion as pervasive environments are characterized by a rather dynamic behavior including mobility and context switches.

The asynchronous and data driven nature of the pub/sub paradigm prevents an application to make assumptions about the time it will take before notifications required for bootstrapping are published. For instance, when a mobile client spontaneously appears in a new location, it cannot rely on notifications being published as soon as it enters. In this kind of scenario, system responsiveness is not only degraded, but the time window in which a client application can actively "listen" is naturally constrained by the duration it stays at this particular location.

1.3 Proposed Approach

This work concentrates in enhancing the mobile pub/sub middleware by reducing the bootstrapping latency. Based on the assumption that a consumer is initialized by a sequence of notifications, recently published notifications are delivered to consumers on bootstrapping phase. These notifications are used to minimize the bootstrapping latency as showed in Figure 1.2. As an example we go back to the currency exchange rate scenario: here, some users could suffice themselves with the *last value*, while considering that the rate may have varied a few points. Other client applications like the one used to generate a chart as a function of the last published values need more than one notification, regardless if they were created yesterday, as long as they are the last.

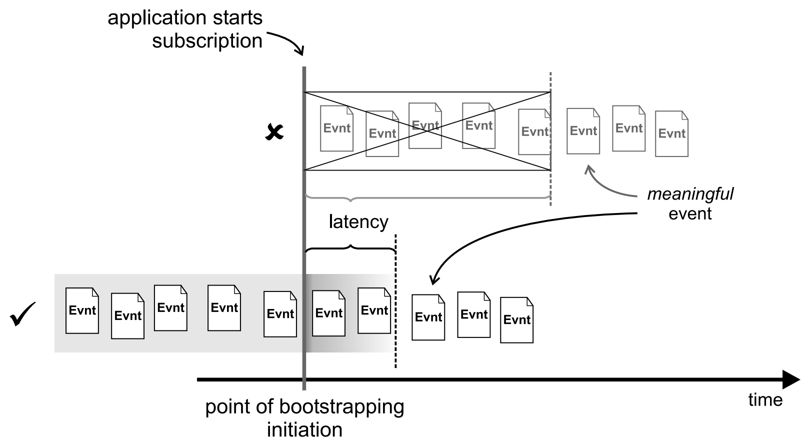

application starts subscription
✗
Evnt
meaningful event
latency
✓
point of bootstrapping initiation
time

Figure 1.2: Adaptation of the bootstrapping phase.

This new quality of service (QoS) can be achieved by extending the pub/sub system to store recently published notifications in its infrastructure. Particularly in this thesis we enhance an already developed pub/sub notification service (described later in more detail in section 2.5). Hence, an important goal is to minimize the amount of changes on the underlying system code. As we will see, the notification service is composed of a network of cooperative brokers. Thus integrating the caching ability and querying those caches is not straightforward.

An adequate way to support dynamic scenarios (i.e., fixed distributed systems with high subscriptions and unsubscriptions rate and/or mobile computing) is to store the notifications in caches distributed in the network. This approach effectively offers a way to integrate data repositories into the network. There are several ways to integrate and query these repositories. This work explores and analyzes some alternatives and strategies and also provides a detailed practical evaluation of them.

1.4 Goals and Contributions of this Thesis

The focus of this thesis is on enhancing mobile publish/subscribe notification services in order to reduce the bootstrapping latency that information-driven applications suffer. The foundation of this work is a publication of A. Buchmann, M. Cilia, L. Fiege, C. Haul and A. Zeidler (the reader is referred to [12]), which sketched the first steps of the present thesis. The work has been divided up into a theoretical and a practical part.

The theoretical part deals with the reasoning of feasible caching strategies that might be used within a notification service deployment. Caching is a fundamental concept and widely known technique and it is applied in many areas like Operating Systems, Databases, etc. It is used, for instance, to speedup page translations, memory locations, file blocks or network routes. On the other hand, event-based systems are no longer only a research field: there are commercial applications in use since the last two decades. Nevertheless, we are not aware of event-based infrastructures which provide support for mobile pub/sub applications facing the described problem. Therefore, in this work this problematic is carefully investigated and developed. Finally, recommendations are pointed out on how to select and combine the caching strategy with its main counterpart, that as we will see, is the routing algorithm.

The practical part describes the implementation that supports the caching functionality as an add-on of the Rebeca notification service. The explicit contribution in this regard is the specification, design, implementation and corresponding evaluation of a model that allows several strategies to be used.

1.5 Issues not Addressed in this Thesis

First, we make a clear distinction between *devices'* mobility and *code* mobility. The latter is an innovative approach based on the ability to migrate code (instead of data) across the nodes of a network, in order to provide a higher level of configurability, scalability, and customizability to the distributed applications [17]. As an emerging research field, code mobility is generating a growing body of scientific literature and industrial developments. We concentrate on mobility of devices as a research issue, while the latter (*code* mobility) is out of the scope of this thesis.

The second issue not addressed is the support for mobile information production. As described later in Section 3.1, this could be handled with the Sensor Networks approach. Wireless Sensor Networks (WSN) are envisioned to fulfill complex monitoring tasks in the near future. A typical WSN application like object tracking fuses sensor readings produced by nodes throughout the network to obtain a high-level sensing result such as the current speed of a tracked vehicle.

1.6 Structure of the Thesis

The rest of the thesis is organized as follows. In **Chapter 2**, a background on the building blocks of this work are described. Of great interest is the introduction to information-driven applications, as well as the infrastructure necessary to support these applications, in general, and the Rebeca notification service, in particular. In **Chapter 3**, the proposed approach is exposed by further classifying the scenarios and analyzing where this work points to. **Chapter 4** starts by dictating the overall requirements that caching strategies must obey, then it highlights the differences between traditional caching and caching in event-based systems. The focus of this chapter is to develop several caching strategies, starting from the simplest possible scheme. In **Chapter 5**, the mapping of these strategies into the design and implementation is pointed out. **Chapter 6** presents a practical evaluation that has been carried out by using the implemented notification service infrastructure enhancements. **Chapter 7** concludes this work and sketches areas of future work.

Chapter 2

Background

In this chapter we intend to provide a background in order to understand the context of the work. The first section starts by introducing the reader into the distributed systems area. Then we move into a more specific issue that is the middleware for mobile computing, which settles a ground to base our reasonings and decisions of the following chapters. Next, we illustrate how several scenarios can be grouped in their own type, so-called information-driven applications. Finally, the publish/subscribe systems are presented as the first choice for implementing such applications. Particularly, the Rebeca notification service architecture is presented as the infrastructure where enhancements will be developed.

2.1 Distributed Systems

A distributed system consists of a collection of components distributed over various computers (also called hosts) connected via a computer network. These components need to interact with each other, in order, for example, to exchange data or to access each other's services. Figure 2.1 illustrates an example of a distributed system.

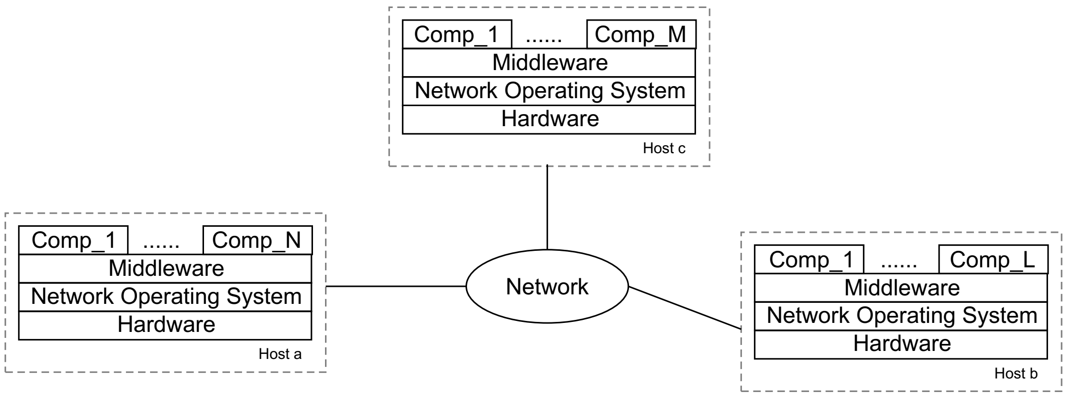


Figure 2.1: Example of a distributed system

Building distributed applications directly on top of the network layer would be extremely tedious and error-prone. Application developers would have to deal explicitly with all the non-functional requirements of these applications. Although this interaction may be built by calling network operating system primitives, this would be too complex for many application developers. Instead, a middleware is layered between distributed system components and network operating system components. *Middleware* implements the *Session* and *Presentation Layer* of the ISO/OSI Reference Model (see Fig. 2.2). Its main goal is to enable communication between distributed components. To do so, it provides application developers with a higher level of abstraction built using the primitives of the network operating system. During the past years, middleware technologies for distributed systems have been built and successfully used in industry, for example, object-oriented technologies like OMG CORBA, Microsoft COM and Sun Java/RMI-IIOP, or message-oriented technologies like IBM MQSeries and TIB/Rendezvous.

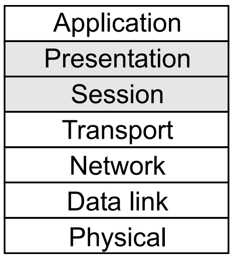


Figure 2.2: The ISO/OSI Reference Model

The previously given definition of distributed system applies to both fixed and mobile systems. Nevertheless, many differences exist between these two that are explored in the following section.

2.2 Middleware for Mobile Computing

Recent advances in wireless networking technologies and the growing success of mobile computing devices such as laptop computers, third generation mobile phones, personal digital assistants (PDAs), watches and the like are enabling whole new classes of applications. However, the requirements of the middleware these applications make use of are very different from those requirements for fixed distributed systems. In order to understand these differences, [26] extrapolated three concepts hidden in the definition of Distributed Systems that greatly influence how to implement middleware for mobile computing: the concept of *device*, of *network connection* and of *execution context*. This comparison will then allow us to reason on solutions to our problem on both environments.

- *Type of Devices:*

  Fixed: Varying from home PCs, to Unix workstations, to mainframes: Generally powerful machines, with large amount of memories and very fast processors.

  Mobile: Varying from PDAs, to mobile phones, to smartcards: limited capabilities, like slow CPU speed, scarce memory, low battery power and small screen size.

- *Type of Network Connection:*

  Fixed: Usually permanently connected to the network through continuous high-bandwidth links. Disconnections are either explicitly performed for administrative reasons or are caused by unpredictable failures. These failures are considered sporadic and therefore treated as exceptions to the normal behavior of the system.

  Mobile: Connectivity to the Internet is possible via wireless local area network protocols, such as WaveLAN. These protocols may achieve reasonable bandwidth if the hosts are within reach of a few hundred meters from their base station. But as all different hosts in a cell share the bandwidth, when they grow in quantity, the bandwidth rapidly drops. Moreover, if the hand-held devices are moved to an area with no coverage or with high interference, bandwidth may suddenly drop to zero and the connection may be lost. Unpredictable disconnections cannot be considered as an exception any longer, but they rather become part of normal wireless communication. Wide area wireless network protocols, such as GSM, have a broader coverage but provide bandwidth that is smaller by order of magnitude than the one provided by fixed network protocols (e.g., 9,600 baud against 100Mbps). Also, GSM charges the users for the period of time they are connected; this pushes users to patterns of short time connections. Either because of failures or because of explicit disconnections, the network connection of mobile distributed systems is typically intermittent.

- *Type of Execution Context:* With context, we mean everything that can influence the behavior of an application; this includes resources internal to the device, like amount of memory, screen size, etc., and external resources, like bandwidth, quality of the network connection, *location*, nearby hosts (or services), and so on.

  Fixed: More or less *static*. Bandwidth is high and continuous, location almost never changes, hosts can be added, deleted or moved, but this happens infrequently as they are big, heavy and expensive. Services may change as well, but the discovery of available services is easily performed by forcing service providers to register with a well-known location service.

  Mobile: Highly *dynamic*. Hosts may come and leave rapidly, and the services that are available when clients disconnect from the network may not be there anymore when they reconnect. Service lookup is much more complicated in the mobile scenario, specially if there's a complete lack of a fixed infrastructure This is why clients cannot assume any knowledge about the configuration of the context, particularly the identity of any default server to contact for a service lookup operation. *Location* is no longer fixed: the size of wireless devices has shrunk so much that most of them can be carried in a pocket and moved around easily. Depending on where the clients are and if they're moving, bandwidth and quality of the network connection could vary greatly as well.

2.3 Information-driven Applications

Although the architecture of current large-scale, networked computer systems is dominated by synchronous client/server platforms (e.g., the World Wide Web, CORBA, J2EE, and COM+), a large range of applications cannot be realized efficiently by using the request/reply paradigm. For instance, consider the scenario where an automated stock trading program monitors stocks and bases its decisions to sell or buy certain stocks on current real-time quotes. If this program is realized using request/reply, it has to periodically retrieve the current quote of a specific stock by requesting it from a quoting server. Each time communication and processing cost would be incurred, and in many cases no new information is delivered. This approach is called *polling*. Polling leads to resource waste because it unnecessarily saturates the servers, the network, and the clients. This not only impedes scalability, in a large-scale system, it may even lead to a total breakdown of the network connections or the server itself. Polling is also inappropriate for applications that run on mobile devices (e.g., PDAs). These devices are especially susceptible to resource waste because of their limited processing power and network bandwidth.

In this example, it is important that quote updates are available to the trading program with *minimum latency* and that no quote updates are missed. Both requirements can only be met using a high polling frequency. Hence, data freshness and completeness are inconsistent with a low data fetching overhead. Moreover, synchronous polling blocks the client until the reply arrives. This means that multiple polling requests (e.g., for different stocks) either have to be executed sequentially or error-prone multi-threaded polling must be used. The same would be true if multiple data sources (e.g., different stock exchanges) were involved. As another examples we can enumerate:

- Enterprise Applications which keep the user/application up-to-date with

  Auction processes' current state

  Stock Quote values

  Real-Estate offerings

  Currency Exchange rates

- Driver support

  Malfunctions of my car

  Traffic jams

  Nearby Gas Stations

  Parking spaces

- Infotainment support

  Coming TV shows related to Snakes, Crocodiles, etc.

  Weekend activities in my city/region

- News Delivery, i.e.

  Results of NBA → LA Lakers playoffs

  Terrorism or suicide attacks

- Travel/Turist support

  Nearest Tourist Info Center

  Nearby Hotels

These examples are not in any way exceptions but rather examples of typical *information-driven* applications, in which information provided by a service depends on information supplied by other services. Realizing such applications using request/reply will always lead to implementations that do not scale and provide data that is inaccurate and probably incomplete. These problems are approached by a new communication paradigm called *publish/subscribe* (pub/sub) that recently gained increased publicity in the distributed systems research area.

2.4 Publish/Subscribe Systems

A pub/sub system consists of a set of clients that asynchronously exchange notifications, decoupled by a notification service that is interposed between them. Clients can be characterized as producers or consumers. Producers publish notifications, and consumers subscribe to notifications by issuing subscriptions, which are essentially stateless message filters. Consumers can have multiple active subscriptions, and after a client has issued a subscription the notification service delivers all future matching notifications that are published by any producer until the client cancels the respective subscription. Data is proactively disseminated (to interested consumers) when it is produced placing in that way a continuous query on produced information. In this thesis it is assumed that producers and consumers are implemented in a way such that they publish and subscribe to the intended events as required by the application.

Publish/subscribe systems have a number of interesting characteristics, as we enumerate next:

1. Producers do not need to directly address consumers and vice versa. Instead, consumers simply specify the notifications they are interested in. This loosely coupled approach facilitates flexibility and extensibility because new consumers and producers can be added, moved, or removed easily.

2. The quantity of producers and consumers can be dynamically determined. This enhances the flexibility by allowing clients to evolve without disrupting the existing system.

3. Communication is asynchronous, thereby removing the disadvantages and inflexibility of synchronous communication previously mentioned.

4. Producers and consumers do not need to be available at the same time. This means that a subscription causes notifications to be delivered even if producers join after the subscription was issued.

5. Publish/Subscribe directly reflects the intrinsic behavior of information-driven applications because communication is initiated by producers of information.

The benefits of pub/sub make them first choice for implementing information-driven applications. For example, publish/subscribe is well suited for information dissemination applications like news delivery, stock quoting, air traffic control [25], and dissemination of the state of auction processes [8]. The use of publish/subscribe techniques has also been described in the areas of mobile agents [33], workflow systems [13], ubiquitous computing [24], peer-to-peer systems [20], and process control systems [22]. Its use was even proposed for loose coupling of components [7] or several independent distributed applications.

2.4.1 System Structure

In order to start understanding the proposed enhancements, a traditional pub/sub system is characterized. The basic event system's architecture is depicted in Figure 2.3. According to [6], it corresponds to an independent processes, data-flow architectural style. The components shown consist of a message manager or *notification service* (`aPublishSubscribeSystem`) and clients. These in turn can indistinctively be subscribers of the information (`aConsumer`), publishers (`aProducer`), or both at the same time (`aConsumerNProducer`). The connectors between these components are the communication links between them. This scheme's dynamic behavior allows the following interactions: A consumer registers its interests by *subscribing* itself with the notification service, using some expression as a message *Filter*. When a producer *publishes* a new *Event*, the notification service forwards the message to the interested clients by *notifying* them. Finally, when a consumer is no longer interested in being notified about a subscription he has registered, he can *unsubscribe*.

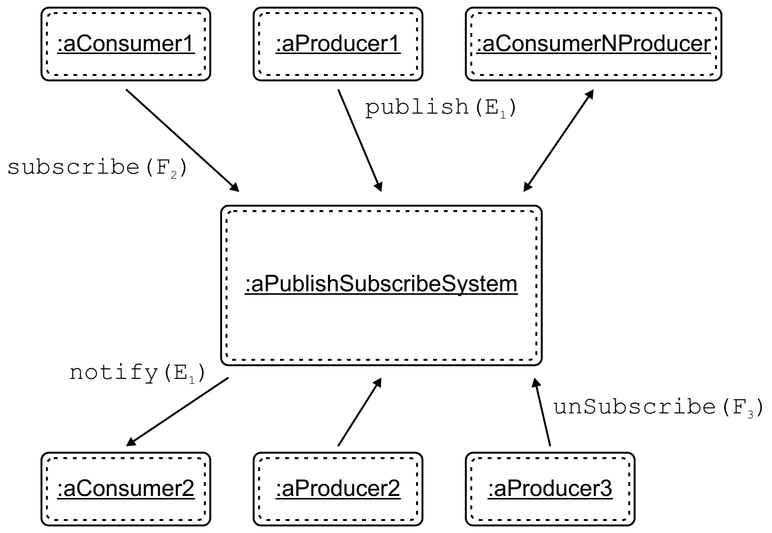


Figure 2.3: Publish/Subscribe systems architecture

The requirements can be demanded for a pub/sub system as follows:

- notifications should be delivered only to clients who have at least one matching active subscription.

- all notifications matching an active subscription should be delivered to the respective client.

- each notification should be delivered to a client *at most once*.

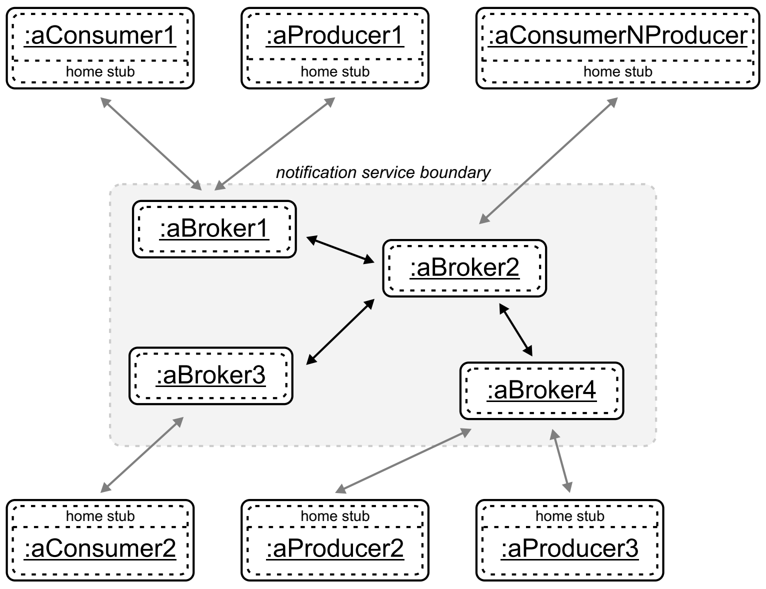


Figure 2.4: Large Scale Publish/Subscribe systems architecture

- notifications should be delivered in some order with respect to their publication (particularly, in a per-producer FIFO order basis).

Figure 2.3 gives a conceptual view or *black box* view of the interface of a pub/sub system. Nevertheless, in practice, scalable implementations of large pub/sub notification services are *distributed*. Hence, in a more detailed, *white box* view of the system structure, the pub/sub component of Figure 2.3 is decomposed into a network of decentralized *brokers*. These brokers conform a (connected) graph, as shown in Figure 2.4. The network's brokers route the subscriptions, unsubscriptions and events from the producers, across a set of brokers, and into the interested consumers of the information using one of several different routing algorithms (which are explained in 2.5.2). Further, each client of the notification service is composed of the application itself and a *stub* that provides a local interface to the pub/sub system where to delegate the operations that the notification service offers. This stub acts as a client-side proxy and is provided by the infrastructure itself in the form of a class library.

2.4.2 Addressing Models

Publish/Subscribe naturally decouples producers and consumers of messages. This is achieved by providing interaction mechanisms that hides physical locations (i.e., network address, socket number, server identity). In this section we describe the existing options to address messages, as well as known related software products that implement these options.

The option adopted by the first generation of pub/sub systems is called *Channel-based* addressing model. A *channel* is used to communicate producers with consumers. Messages were published to a specific channel by producers and delivered to all consumers that had subscribed to it. Channels allow efficient data delivery, but the subscription expressiveness is limited. In the illustration of Fig. 2.5, an example where sport news are delivered from producers to consumers by means of a channel is shown. In CORBA, a primitive event service was introduced to provide a mechanism for asynchronous interaction between CORBA objects. Here, an event channel acts as a mediator between suppliers and consumers of events. To overcome deficiencies of this service specification, the notification service [30] was proposed as a major extension with support for quality of service specifications and basic event filtering.

In the second alternative a *subject* is associated to each message [31]. Subject names consists of one or more elements (usually a string) organized in a tree by a means of a dot notation. This

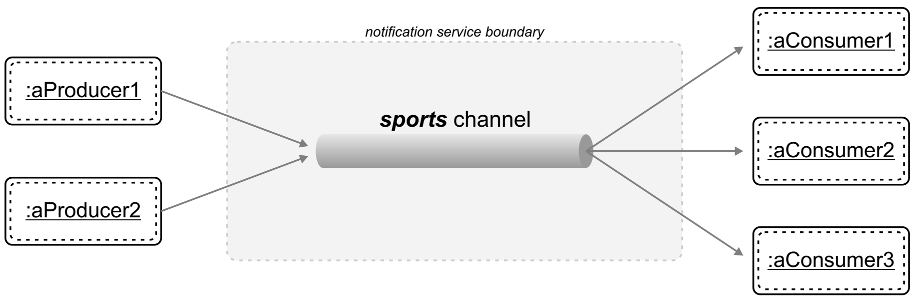


Figure 2.5: Channel-based Addressing

model is denominated *Subject-based* addressing, and features a set of rules that defines a uniform name space for messages and their destinations. This approach is inflexible if changes to the subject organization are required frequently, implying fixes in all participant applications.

The Java Message Service (JMS) provides the Java technology platform with the ability to process asynchronous messages. JMS was originally developed to provide a common Java interface (API) to legacy Message Oriented Middleware (MOM) products like IBM MQ-Series (called today WebSphere MQ) or TIB/Rendezvous. In this way, the JMS API provides portability of Java code allowing the underlying messaging service to be replaced without affecting existing code. JMS provides two models for messaging among clients: *point-to-point*, which corresponds to Channel-based addressing using queues; and *publish/subscribe*, which corresponds to Subject-based addressing using *topics* with some enhancements. Under the topic model, consumers not only subscribe to a channel but can also specify additional boolean predicates by means of a restricted set of SQL `WHERE` expressions [19].

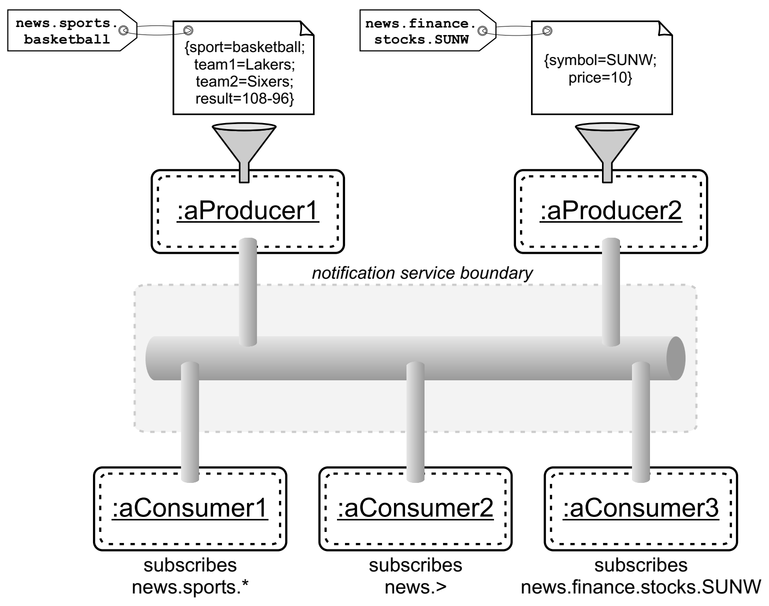


Figure 2.6: Subject-based Addressing

In Figure 2.6, two producers are shown while they're publishing messages. Notice that each message has its corresponding subject, symbolized here using tags/labels. Subscriptions are carried out using the subject name space organization where wildcards can be used to specify consumers interests. For instance, `aConsumer1` subscribes to all sports news (by means of `news.sports.*`); `aConsumer2`, to all kinds of news (`news.>`); and `aConsumer3` subscribes specifically to the stock prices of SUNW (`news.finance.stocks.SUNW`). After producers publish their messages (as depicted in the figure), `aConsumer1` receives one notification with the result of a basketball game, `aConsumer3` receives a notification related to the stock price of SUNW, and `aConsumer2` receives both notifications.

The third approach was proposed in order to improve the expressiveness of the subscription model. The *Content-based* addressing model allows subscriptions to evaluate the whole content of notifications, thus it provides a more powerful and flexible notification selection than Channel- or Subject-based mechanisms, for which the actual content of a notification is opaque. The increase in expressiveness allows the delivery of uninteresting notifications to be reduced or even to be avoided. In particular, this is important for applications that run on mobile devices having limited processing power and network bandwidth. This approach is more flexible, but requires a more complex infrastructure [9, 10, 28]. Many projects in this category concentrate on scalability issues in wide-area networks and on efficient algorithms and techniques for matching and routing notifications to reduce network traffic. Most of these approaches use simple boolean expressions as subscription patterns since more powerful expressions cannot be treated.

The fourth option, *Concept-based* addressing model appeared because of the need to understand events beyond the closed confines of a single component, application or network. Traditional pub/sub mechanisms only expose the data structure of events (data to be exchanged) to the participants. Event consumers must base on this scarce information to express their interest without having a concrete definition of meaning nor explicit assumptions made by event/data producers. Without this kind of information event producers and consumers are expected to fully comply with implicit assumptions made by participating software components or applications. Even in the cases of a very small set of applications within an enterprise this approach is questionable.

In order to provide a higher level of abstraction to describe the interests of event producers and consumers, this model supports from the ground up ontologies which provide the base for correct data and event interpretation[11, 8]. Rather than requiring every producer or consumer to use the same homogeneous namespace (as is common in other pub/sub systems), it provides metadata and conversion functions to map from one context to another. This last feature allows event consumers to simply specify the context to which events need to be converted before they are delivered for client processing.

In all the options listed above, producers do not have any knowledge of who or what applications are subscribing. Additionally, the physical location of message consumers becomes entirely transparent without requiring a naming service (a.k.a. location transparency).

2.4.3 Publish/Subscribe Systems with Advertisements

Many implementations of publish/subscribe systems have the notion of advertisements which are issued by producers to indicate their intention to publish certain kinds of notifications. Today, advertisements are used for two main reasons: First, as we will see in section 2.5.2, they are applied to optimize implementations of publish/subscribe systems. Second, consumers may want to inspect the advertisements currently available, for example, in order to issue, change, or cancel subscriptions. Besides this, advertisements should also be used to control the notifications a producer publishes. For example, if a notification is published by a client that does not match any of its active advertisements, it should be discarded and not delivered to any client.

2.5 The Rebeca Notification Service

The ideas presented in this work were applied to an existing notification service, Rebeca. Therefore, from now on we proceed by exploring more specifically its architecture and modular separation of the previously mentioned responsibilities.

2.5.1 Overview

REBECA[1] [28] stands for Rebeca Event Based Electronic Commerce Architecture. Briefly, it's an object-oriented notification service framework implemented with JAVA. It was first designed as a prototype with the purpose of analyzing different routing strategies for Content-based pub/sub, along with the usage of advertisements. Then, it was used by several researchers to test different ideas, hence, it is composed of several class packages responsible of different concerns. In [1] an approach is explored in order to extend REBECA to support Concept-based addressing.

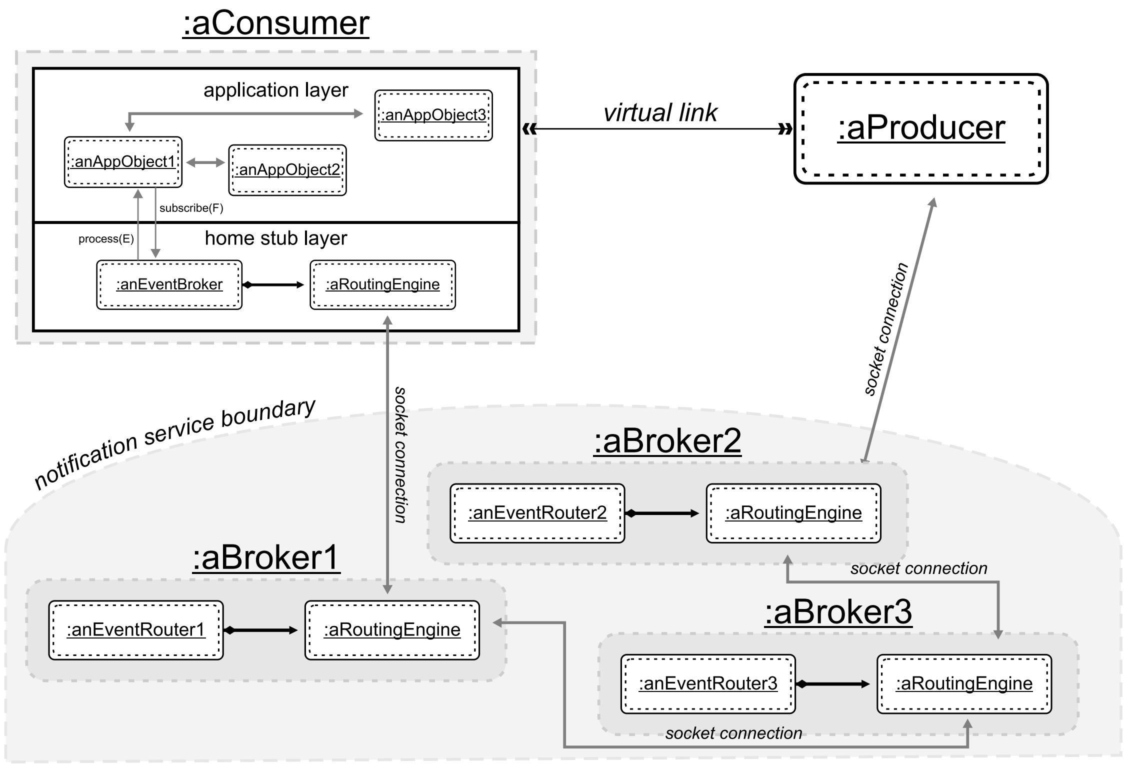


Figure 2.7: REBECA architecture details

Nevertheless, the components we're interested in are those involved in the client-broker interaction. Thus they must be further explored in order to gain insights of it. In Figure 2.7[2], several details arise:

- The client's application layer is composed of several cooperating objects (the arrangement of these objects is not of our concern, though).
- The client's home stub layer is mainly composed of an `EventBroker`. An `EventBroker` serves as the client's basic interface to interact with the notification service. Subclasses of it implement this interface in different ways. Particularly, a `TCPEventBroker` implements this interface through the usage of Sockets and ServerSockets in order to connect to a remote `EventRouter`. An `EventBroker` handles the routing of events by delegating this task to a `RoutingEngine` instance (actually, a subclass of it, like `Flooding`).
- A `RoutingEngine` keeps track of subscriptions and advertisements together with their origin and destinations using a `RoutingTable` to store this information. Events are added through a queue and delivered to each client's destination.
- The network's brokers are instances of the `EventRouter` class. An `EventRouter` is a per host manager of the event system infrastructure. It starts a ServerSocket to listen and accept

[1]See `www.gkec.informatik.tu-darmstadt.de/rebeca`

[2]In REBECA, the `notify(Event)` method's name turns to be `process(Event)`

new connections from `EventBrokers` and other `EventRouters`. Furthermore, currently only TCP/IP connections are established. An `EventRouter` also handles the routing of events delegating this task to a `RoutingEngine` instance.

- The connectors between `EventBrokers` and `EventRouters`, as well as between the `EventRouters` themselves, represent a socket communication.

To specify the behavior as a set of action sequences and represent the functional requirements of Rebeca, a conventional UML Use Case diagram is provided (Figure 2.8). The actors involved are the possible system users: Producer and Consumer. There are six use cases presented; their basic flow is as follows:

1. A Producer *AdvertisesInformation*. This use case starts when the actor contacts the notification service to make public (advertise) in advance the characteristics (or pattern) of events he will publish. He creates an `Advertisement` based on a particular `Filter` that represents the kind of information to publish, and tells its `EventBroker`'s to *advertise* it.

2. A Producer *UnadvertisesInformation*. This use case starts when the actor decides he no longer wants to produce events that previously did. He creates an `UnAdvertisement` based on the previously advertised `Filter` and tells its `EventBroker` to *unadvertise* it.

3. A Producer *PublishesNotifications*. This use case starts when the actor has produced new information and wants to publish it. He creates a new `Event` and tells its `EventBroker` to *publish* it. There's a note with a precondition stating that the published notification must match with a previous Advertisement submitted by the producer.

4. A Consumer *GetsNotified*. This use case starts when a producer publishes a notification that this consumer is interested in. The notification service forwards to the client's `EventBroker` the new `Event`. The `EventBroker` in turn *notifies* the Consumer. This use case is *included*[3] by the use case *PublishesNotifications*, and shown using a stereotype. There's a note with a precondition stating that the received notification must match at least one of the consumer's active subscriptions.

5. A Consumer *SubscribesToInterest*. The use case starts when the actor decides he wants to receive new kinds of events. He creates a `Subscription` based on a particular `Filter`, which represents the kind of information to get notified on, and tells its `EventBroker` to *subscribe* it. There's a note which reminds that given that most event-based applications have an initial bootstrap phase, the system should deliver certain number of notifications in a short time allowing the application to quickly start working properly.

6. A Consumer *UnsubscribesFromInterest*. The use case starts when the actor decides he no longer wants to receive some types of events. Based on a previously issued `Subscription`, he tells its `EventBroker` to *unsubscribe* it.

The fifth use case's note vaguely describes a quality of the system: is a desire of the applications to *minimize* the time taken to quickly start working properly. The notification service doesn't know how to *minimize* this time. But, if the client specified what does he needs to correctly start interpreting the current flow of notifications properly (to bootstrap), it could do *something*. We will get back to this point in section 3.2, where the proposed approach (previously indicated in section 1.3) is analyzed in more detail.

[3]An include relationship between use cases means that the use case explicitly incorporates the behavior of another use case at a location specified in the base. The included use case never stands alone

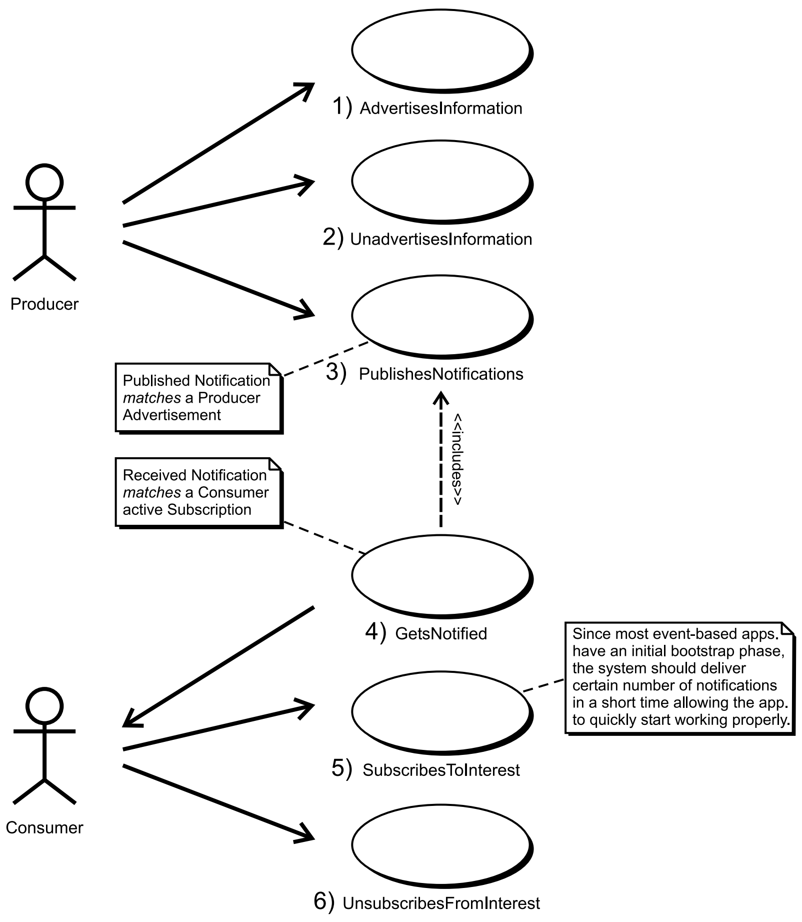

1) AdvertisesInformation
2) UnadvertisesInformation
3) PublishesNotifications
4) GetsNotified
5) SubscribesToInterest
6) UnsubscribesFromInterest
Producer
Consumer
<<includes>>
Published Notification *matches* a Producer Advertisement
Received Notification *matches* a Consumer active Subscription
Since most event-based apps. have an initial bootstrap phase, the system should deliver certain number of notifications in a short time allowing the app. to quickly start working properly.

Figure 2.8: UML use case diagram

2.5.2 Routing Algorithms

As it was shown in Section 2.4.1, the functionality of the notification service was distributed in order to provide a more scalable and fault-tolerant system. This system is thus composed of a set of brokers who manage a subset of the clients, and propagates notifications from producers to consumers along a path of interconnected brokers. To achieve this, each broker forwards a notification it processes to a subset of the brokers it is connected to, i.e., its neighbors. This is done in a way that guarantees that a notification is delivered to all interested consumers. REBECA is restricted to acyclic topologies.

The REBECA notification service has a routing algorithm framework which allowed the following routing algorithms to be implemented. The simplest way to implement a distributed pub/sub system is by *flooding* notifications. This implies that any published notification is processed by every broker. Although it is well suited to systems in which subscriptions are changing at a very high rate, a lot of notifications may be forwarded unnecessarily because each notification is sent to every broker (regardless of whether or not it has a local client with a matching subscription).

This drawback led to the idea of filter-based routing. Here, each broker has a routing table that is used to route notifications based on their content towards interested consumers. Compared with flooding, filter-based routing reduces the quantity of notifications that are forwarded, but complicates notification forwarding and introduces the necessity to update routing tables if subscriptions change or if new subscriptions are issued.

With *Simple Routing*, new and cancelled subscriptions are simply flooded into the broker network such that they reach every broker. Hence, each broker knows *when* to forward a newly received notification, as well as *where* to do so in such case. This strategy is simple but it implies that *every* broker has global knowledge about *all* active subscriptions (*every* routing table contains a routing entry for *every* active subscription).

By taking into account similarities among the subscriptions, global knowledge of active subscriptions is avoided. In this way, the notification service performance and scalability is enhanced (for concrete results the reader is referred to [29]). The following strategies do so but differ in the resulting routing table size.

When using *Identity Routing*, a subscription is not forwarded to a neighbor if an *identical* subscription (that has not been cancelled) was forwarded to that neighbor. Two filters are said to be identical if they match exactly the same set of notifications.

Another strategy is *Covering Routing.* A filter *covers* another filter if the former matches at least all notifications the latter matches. A subscription is not forwarded to a neighbor if another subscription (that has not been cancelled) that covers the former was forwarded to that neighbor.

A different approach is *Merging Routing.* A broker can *merge* the filters of existing routing entries and forward this merger to a subset of its neighbors. The generated mergers are forwarded in a way such that only interesting notifications are delivered to a broker.

Routing with Advertisements implies that advertisements are issued by producers to indicate their intention to publish certain kinds of notifications. Similar to subscriptions, each client can have multiple advertisements which are cancelled separately. Advertisements can be used as an additional mechanism to further optimize content-based notification routing. For this purpose a second routing table is managed by every broker. This advertisement routing table is maintained by the same algorithms as the subscription routing table, i.e., by forwarding new and cancelled advertisements through the broker network. While the subscription routing tables are used to route notifications from producers to consumers, the advertisement routing tables are used to route subscriptions from consumers to producers: a subscription is only forwarded to a neighbor if it overlaps with an active advertisement that has been received from this neighbor before. Most filtering-based routing algorithms can be combined with advertisements.

2.6 Summary

This chapter presented the essential blocks used to build up the present work. In Section 2.1 we saw the composition of a distributed system. Distributed systems techniques have attracted much interest in recent years due to the proliferation of the Web and other Internet-based systems and services. Moreover, people have an increasing desire for ubiquitous access to information anywhere, anyplace and anytime. Hence they need not only mobile and portable devices but adequate communication systems and infrastructures that differ with the conventional (i.e., fixed) systems. The distinction between the two was made clear in Section 2.2.

Application developers will benefit designing information-driven applications (Section 2.3) with the publish/subscribe paradigm because of their characteristics (enumerated in Section 2.4.1). Finally, in Section 2.5, the shape of a large-scale, content-based, publish/subscribe system called Rebeca was described. With this description the reader is aided in the understanding of the guts of the system over which the enhancements were developed.

Chapter 3

Proposed Approach

In this chapter we explore the scenarios presented previously in Section 2.3. The scenarios will help the reader to locate in the context of the possible applications this middleware is intended for. Then, we capture the intended behavior of the enhancement and how to support interchangeable caching strategies. We abstract the caching strategies' generic behavior, which serves as an introduction for the next chapter.

3.1 Scenario Classification

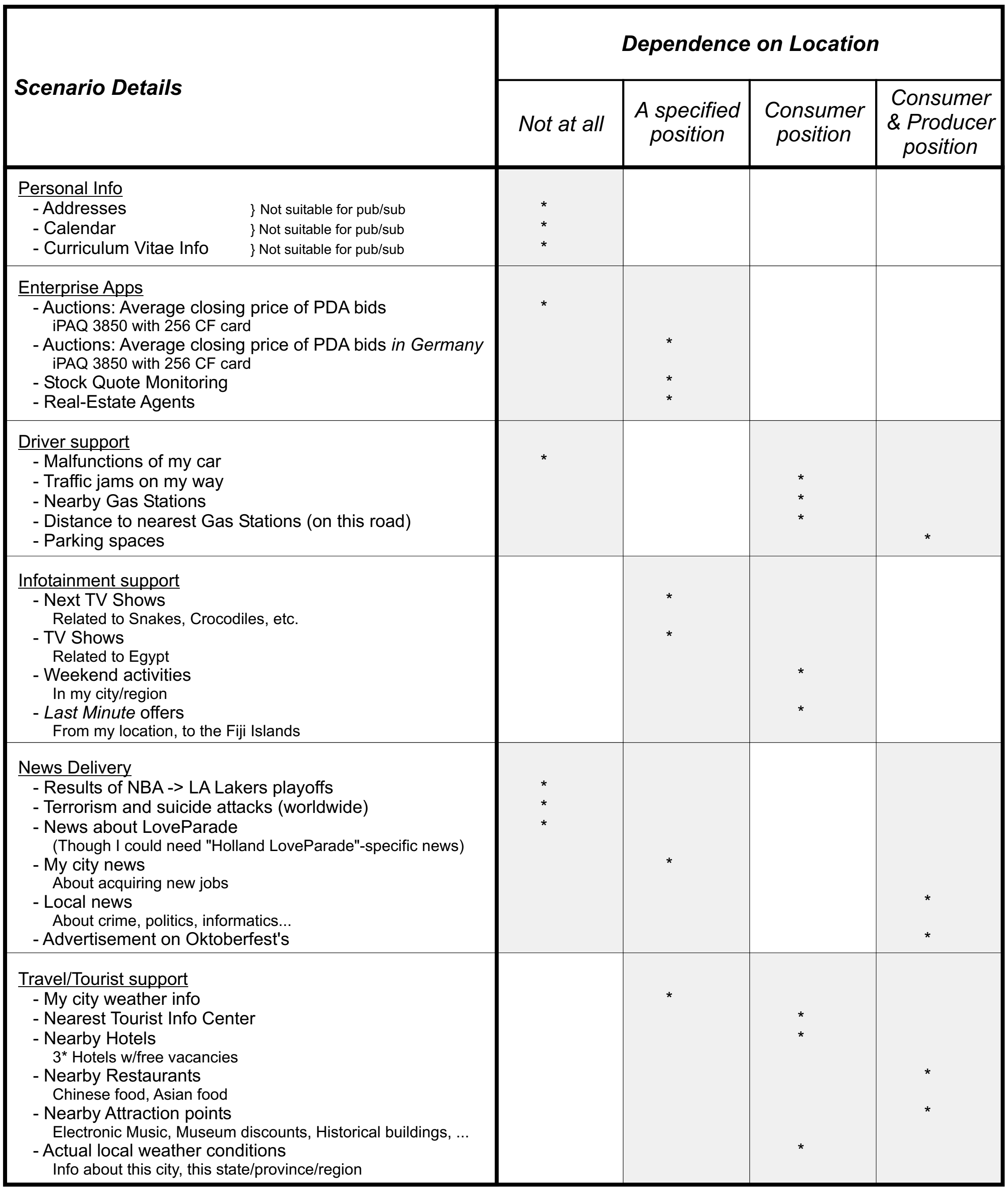
The pub/sub middleware can be used to implement information-driven applications like the examples listed in Section 2.3. Here, these traditional messaging scenarios will be further explored in order to have a brief description of the range of environments where some strategies behave better than others.

Not all of these scenarios can be realized with the current implementation of Rebeca. In order to check how Rebeca adapts to the requirements of the previously enumerated scenarios, the list was extended and classified. These scenarios were classified according to the consumer or producer of information's mobility. This classification will allow us to see which of them can already be implemented (Figure 3.1). The scenarios' mobility was splitted into four classes (and mapped to the four respective columns):

1. These scenarios are not location-dependent at all, in the sense that the subscriptions and events are not related to a location.
2. These scenarios somehow include the notion of locations, but a specific, concrete one. Here it doesn't care where the information is to be consumed or where was it generated, it only cares what the information says that it is about.
3. These scenarios consider information from fixed locations, but the consumer's location is important to see if notifications match a filter or not.
4. These scenarios consider information from moving targets and the consumer's location, to see if notifications match a filter or not.

We can state that while the scenarios in the first column can be realized with the current implementation, the others don't. The third and fourth columns depend both on client and information (possibly varying) location, hence an extra mechanism is needed in order to provide the information of context changes. This could be handled with the *Wireless Sensor Networks* approach, which is a technology envisioned to fulfill complex monitoring tasks in the near future, though this is not an addressed issue in this thesis.

However, scenarios in the second column can be implemented using Rebeca with the Concept-based addressing model. This model could be extended with representations for the space or locations, and provide a minimal set of operations, particularly with a semantic metadata model like MIX which allows information exchange in loosely coupled environments like pub/sub systems.



Scenario Details	***Dependence on Location***			
	Not at all	*A specified position*	*Consumer position*	*Consumer & Producer position*
Personal Info				
- Addresses } Not suitable for pub/sub	*			
- Calendar } Not suitable for pub/sub	*			
- Curriculum Vitae Info } Not suitable for pub/sub	*			
Enterprise Apps				
- Auctions: Average closing price of PDA bids iPAQ 3850 with 256 CF card	*			
- Auctions: Average closing price of PDA bids *in Germany* iPAQ 3850 with 256 CF card		*		
- Stock Quote Monitoring		*		
- Real-Estate Agents		*		
Driver support				
- Malfunctions of my car	*			
- Traffic jams on my way			*	
- Nearby Gas Stations			*	
- Distance to nearest Gas Stations (on this road)			*	
- Parking spaces				*
Infotainment support				
- Next TV Shows Related to Snakes, Crocodiles, etc.		*		
- TV Shows Related to Egypt		*		
- Weekend activities In my city/region			*	
- *Last Minute* offers From my location, to the Fiji Islands			*	
News Delivery				
- Results of NBA -> LA Lakers playoffs	*			
- Terrorism and suicide attacks (worldwide)	*			
- News about LoveParade (Though I could need "Holland LoveParade"-specific news)	*			
- My city news About acquiring new jobs		*		
- Local news About crime, politics, informatics...				*
- Advertisement on Oktoberfest's				*
Travel/Tourist support				
- My city weather info		*		
- Nearest Tourist Info Center			*	
- Nearby Hotels 3* Hotels w/free vacancies			*	
- Nearby Restaurants Chinese food, Asian food				*
- Nearby Attraction points Electronic Music, Museum discounts, Historical buildings, ...				*
- Actual local weather conditions Info about this city, this state/province/region			*	

Figure 3.1: Scenario classification

3.2 Proposed Approach Analysis

As we stated previously, the proposed approach is based on the storage of notifications in cache queues distributed in the network's brokers. Thus, two issues must be dealt with in order to introduce the proposed changes:

- We must think of some means for the client application to specify *how many (or how old)* notifications does it need to bootstrap (i.e., the external changes).

- We must think of how would the notification service use this information, in order to try to do an effort to provide the required notifications (i.e., the internal changes).

As explained before in section 1.3, this strategy is based on the assumption that applications can be bootstrapped with recently published notifications, which means that client applications make no distinction between new or "recently" published notifications. We've nicknamed this process as *subscribing into the past.* Though normally events have an associated *time to live* attribute, we're dealing with applications willing to pay the cost of not considering it against the gains in response time.

To attend the first issue, we need to clearly specify what does the client application need to bootstrap. Basically, an application could specify its needs considering at least the following options:

- how *many* notifications does it need to bootstrap (i.e., an integer amount),
- how *old* notifications from the past does it need to bootstrap (i.e., related notifications published a specific number of minutes ago), or
- a *mixture* of the previous two (i.e., an integer amount of notifications published within a number of minutes ago).

This *"specification of the application needs"* is called *bound* and serves as a parameter for the strategies to determine which matching notifications to fetch from the past (if any available).

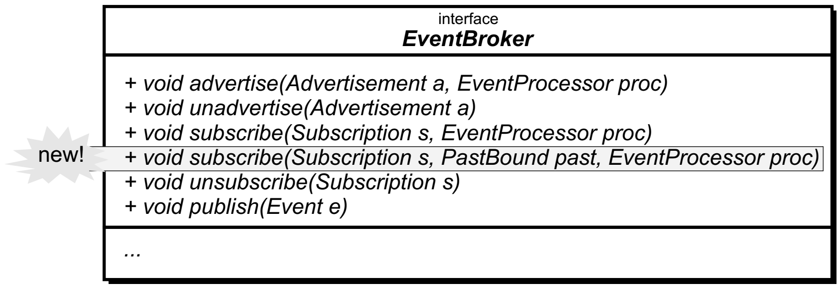

interface
EventBroker
+ void advertise(Advertisement a, EventProcessor proc)
+ void unadvertise(Advertisement a)
+ void subscribe(Subscription s, EventProcessor proc)
new!
+ void subscribe(Subscription s, PastBound past, EventProcessor proc)
+ void unsubscribe(Subscription s)
+ void publish(Event e)
...

Figure 3.2: Extending the pub/sub interface

Then, we need to provide a means to specify this *bound* to the notification service. For this purpose, the pub/sub *subscription* interface needs to be extended. When the client application makes a new subscription, it specifies its interests with a `Subscription` (constructed with a `Filter`), as well as its needs (encapsulated in a `PastBound` object) to the `subscribe()` method (Figure 3.2). In this way the notification service can check to see if it's able to fulfil the user requirement. Also, by *adding* a new `subscribe()` method (instead of changing the old one), we avoid forcing old clients to be changed in order to conform to the modified method's signature.

To attend the second issue, we propose an extended system behavior which is common to every caching strategy. The general idea is to incorporate a component that intervenes in the client subscription process by adding the caching functionality. The system's new generic behavior is illustrated as a specialized UML sequence diagram in Figure 3.3[1]. Given that the UML specification makes no provisions for introducing elements that connect directly to methods, the specialized diagram contains a new element. It is drawn as a circle with a cross inside, which represents the method that must be intercepted. This UML diagram specialization was adapted from the proposal in [5]. There, they don't state how should the sequence description be ordered. However, an ad-hoc ordered description follows:

[1]For simplicity, the diagram shows `aConsumer` subscribing itself (i.e., `this`) to be notified and process incoming events, which could not always be the case

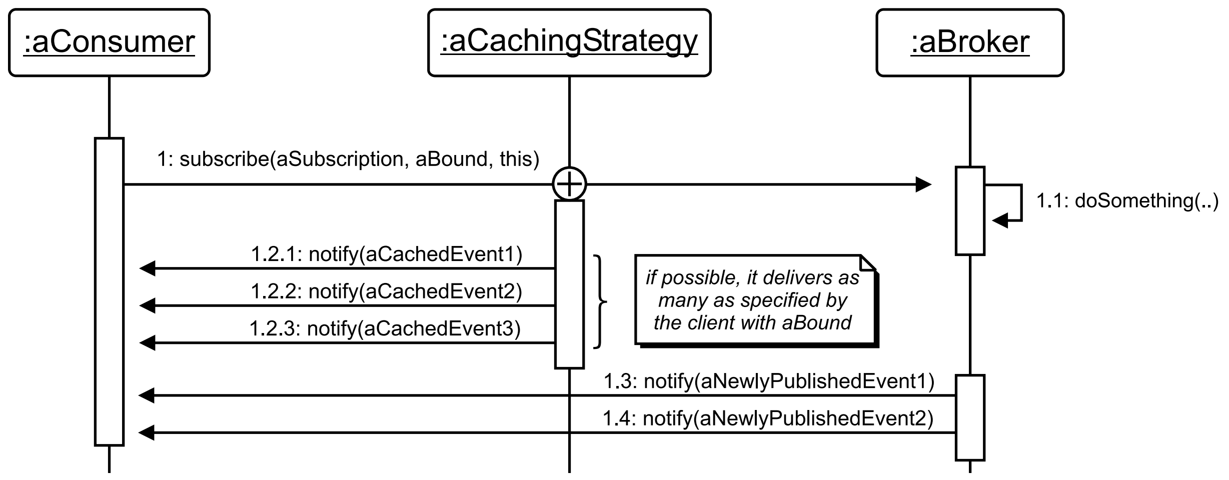

:aConsumer
:aCachingStrategy
:aBroker
1: subscribe(aSubscription, aBound, this)
1.1: doSomething(..)
1.2.1: notify(aCachedEvent1)
1.2.2: notify(aCachedEvent2)
1.2.3: notify(aCachedEvent3)
if possible, it delivers as many as specified by the client with aBound
1.3: notify(aNewlyPublishedEvent1)
1.4: notify(aNewlyPublishedEvent2)

Figure 3.3: New subscribtion behavior

1. First, the client application issues a subscription with a `Subscription` and a `PastBound` as arguments (previously explained).

   1.1 Depending on the actual broker deployment configuration parameters, different internal duties are performed.

   1.2 The subscription invocation is intercepted by a `CachingStrategy`, which queries the caches to find matching notifications (observe the circle with a cross inside).

   1.2.1 - 1.2.3 Just as in the normal operation, the enhanced pub/sub system delivers the required notifications that match the specified filter through the `notify()` callback method. This makes opaque to the consumer that those received notifications have already been delivered *in the past*.

   1.3 - 1.4 After sending the solicited notifications stemming from the past, standard delivery of present and future notifications commence operation.

The notification service will capture this *bound* and query the broker network to fetch the notifications. This generic behavior could be specialized in several ways, which we call *caching strategies*. There might exist several strategies differing in their simplicity, effectiveness of notification fetching, memory overhead, network utilization, data access mechanisms and requirements from the infrastructure. The next chapter gradually explores in detail and provides an adequate analysis of feasible alternatives.

3.3 Summary

In this chapter we have immersed ourselves in the problem by classifying the scenarios where minimizing the bootstrap latency clearly enhances the system's quality of service. Then, we have focused and planned how to face the problem, beginning by the outer changes and ending by a generic behavior which considers, since the early stages, that several caching strategies might exist. As a result, a `Caching` component was introduced into the infrastructure that is responsible of the caching concern. In the following chapter we will describe how this component can help to reduce or eliminate the bootstrapping latency, by carrying out an analysis of the mentioned concern.

Chapter 4

Caching Strategies: Analysis

This chapter provides an analysis of the caching strategies. First, requirements on caching strategies are dictated. Next, a comparative analysis between traditional caching systems and caching in event-based systems is given. Then, caching issues are developed in two sections: data storage and data querying. Since several strategies are developed, we summarize the chapter by providing some guidelines on choosing the appropriate strategy.

4.1 Caching Strategies Requirements

Although there are several caching strategies to implement, we can enumerate a number of overall requirements that they all must fulfil. These requirements are listed as follows:

- *Response Time.* As stated by Stankovic and cited in [4], "the objective of *fast computing* is to minimize the average response time for some group of services". This quality attribute is the central issue driving the strategies development. This also suggests that metrics must be used in order to establish a comparative working framework of the times required by applications to bootstrap.

- *Time overhead*, related to the worst-case overhead (e.g., not enough notifications were published to minimize the bootstrapping sequence). Though the idea here is best effort (we can not guarantee nothing), compared to the traditional case where no strategies are applied at all, the strategy should ensure that the bootstrapping latency is not increased.

- *Space usage*, related to a controlled, judicious and customizable caching. The strategies should not deliberately store everything in order to enhance the QoS (this would convert the system into a complete event history repository, eliminating the transient nature of the notifications). However, it is mandatory to have some means to specify or otherwise limit the broker resources available for a strategy to execute. This could be measured in terms of the per-host cached entries amount.

- *Integrability,* in several forms:

  - *Correctness.* The functional requirements or behavior of the pub/sub system must hold while the strategy is performing. Basically it must preserve the per-producer FIFO order of event delivery at each broker.
  - *Transparency*, by delivering the notifications from the past as with the normal behavior (i.e. using the `notify()` interface). Nevertheless the consumer should not assume (or be aware that) these notifications occurred in the past, since that depends on the internal policies of the caching strategy being in use.

  - *Dynamism*, by structuring the solution to provide a flexible way of selecting the caching strategy that best fits, allowing a *plug and play* feature of this functionality at broker startup-time.

- *Separation of Concerns*, as a means to cleanly separate the essential quality attribute of performance with the rest of qualities of the system (as transactions, security, persistence, etc).

- *Specification of the requirements* that each strategy has from the infrastructure. Requirements as *global ids*, *synchronized clocks*, etc., might not be provided by every notification service. Hence, specifying this kind of requirements allow the strategy to be reused on any notification service that accomplish them.

These requirements might be accomplished directly or not. For instance, the judicious space usage can be managed by arranging a customizable data structure (which is not that complex), whereas designing a flexible mechanism to select the caching strategy is not straightforward.

4.2 Caching in Event-Based Systems

In event-based systems, *caching* is applied in a different fashion than in traditional systems (e.g., Database Management Systems or Operating Systems). Hence it is important to recall these usages and stand out its differences.

Traditionally, caching's concept is the following: information is kept in some storage system (e.g., disk). When this information is needed, it is copied into a faster storage system (i.e., main memory), in a temporary basis. Later, when a piece of information is needed, the cache is first checked. If the piece is found, it is used directly. Otherwise, information is used from the main storage system and a copy is made into the cache under the assumption that this piece of information will later be used. With this in mind, several issues can be stated:

1. *Cache Coherency and Consistency:* In standard systems, caching structures represent intermediate layers which hold (perhaps modified) copies of underlying data. Hence in these environments, updating strategies must be clearly defined (e.g., implicit or explicit). Nevertheless, in event-based systems, these structures hold copies of *read-only* data. These notifications are never sent back to the creator or *publisher* (unless itself subscribes to its own information). Though there might exist several copies of the same notifications spread over the notification service, all of them have exactly the same values hence no consistency problems can arise.

2. *Cache Sizes:* Given that cache sizes is limited, cache management is very important. This is true in both kind of systems.

3. *Information lifetime:* In event-based systems, information is often said to be short-lived. Though, the period of validity of the information managed by standard systems' caches vary from very short (like cached disk sectors) to very long (like cached WWW images).

4. *Buffering:* caching and buffering are two distinct mechanisms. Their main difference is that in a *buffered* system each buffer could hold the unique existing copy of a piece of information in order to cope mainly with device speeds mismatches, whereas in a *cached* system, by definition, each cache just holds a copy on a faster storage of an item that resides elsewhere which allows these copies to be progressively updated and modified in order to improve the overall data access efficiency. These functions can be applied together in certain cases, though (e.g., operating systems disk I/O drivers [32]). In our case, we use caches as notifications buffers (recall that cached events occured *in the past*) with the goal of performance improvement, hence taking the best from each mechanism.

Having defined these similarities and disparities, we proceed deepening the subject by developing two issues which are closely related though different: *Notifications Storage* and *Cache Querying*. The rest of this chapter contains an in-depth analysis of both subjects.

4.3 Notifications Storage

In order to decide *when* and *how* to cache notifications received and (perhaps) forwarded by the brokers, policies must be defined.

Regarding the *"when"* to cache notifications we recall the *locality principle*. This concept, sometimes also called *locality of reference*, is a concept which deals with the process of accessing a single resource multiple times. Particularly, *temporal locality* establishes that a resource that is referenced at one point in time will be referenced again sometime in the near future. Traditional cached systems design exploit temporal locality in order to decide when to cache data and how long to maintain this data in cache (which is called *replacement policies*). Nevertheless, in event-based systems, this decision must be made by the brokers based on the information they handle. Basically, this information consists of:

- Subscriptions and Unsubscriptions
- Advertisements and Unadvertisements
- Notifications

It is important to note that caching decisions heavily depend on the broker network deployment configuration. This configuration parameters affect the information observed by the network's brokers, thus making some brokers receive different information than others. Basically this is because of the usage (or not) of advertisements, as well as the routing algorithm being in use. For instance, if advertisements are not being required, then fewer cache management decisions can be made because a broker doesn't know where to get notifications from (i.e., from which neighbor brokers), thus having to resort to an exhaustive approach. Likewise, as we will see in the next section, the fact of a notification being received and/or forwarded by a broker executing the `Flooding` routing algorithm, is not a reason to assume that this notification is really interesting for any consumer (which fully contrasts with the locality principle).

Because of the previously exposed, we argue that caching strategies depend on the broker deployment configuration. This is the underlying reason why an analysis of the relationship between possible deployment configurations and the caching strategies is needed.

On the other hand, regarding the *"how"* to cache notifications, we proceed by analyzing the per broker buffer data structure next. These structures will retain a set of notifications for subsequent queries, which must return them in the same order they've been queued (i.e., the arrival order). We could think of these notification storages as simple bounded length buffers. Since these structures must behave as FIFO queues, they could be implemented as circular ring buffers, hence providing an implicit FIFO ordering to the element position (see Figure 4.1.a).

The main problem with this approach is that storing every notification in a single circular buffer causes high-frequency notification types to overwhelm low-frequency ones. In other words, the buffer has no control over distinct notification types quantities but an overall control.

A possible solution to this problem is to provide parallel circular ring buffers. These buffers can be indexed by subscription's filters (Figure 4.1.b), which allows a more precise administration of the overall buffer size. With this arrangement we can specify the maximum length of both the filters index and the filter's buffers. The required behavior can be described in terms of the following events:

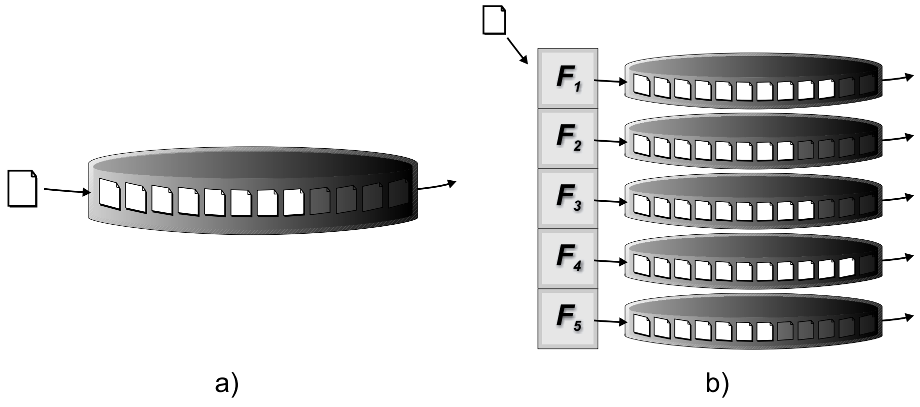


Figure 4.1: Buffering structures

- When a subscription is received, it is assigned an empty index entry and a circular ring buffer is allocated. Since the structure might already contain notifications (stored in other filter's buffers) which match the new filter, the pre-existent filter's buffers are linearly scanned in order to fill the new one.
- When a notification is received, it is matched against every filter. For each matched filter, it is accordingly enqueued.
- Finally, when an unsubscription is received, the respective buffer is disposed.

Although correct for a generic subject- or topic-based addressing notification service, this solution doesn't take advantage of the benefits of a content-based notification service. A more striking idea is to make use of the existent filter relationships such as *identity*, *covering* and *merging.* These relationships can be exploited as described next:

- *Filter identity:* two filters F_1 and F_2 are *identical* (denoted $F_1 \equiv F_2$) iff they match the same set of notifications. Alternatively, we can define filter identity with the function $N(F)$, defined as *the set of all the notifications that match the filter F.* Thus, two filters F_1 and F_2 are identical iff $N(F_1) = N(F_2)$. With this function it is possible to prevent buffer duplicates (in contrast with the previous approach where this was not considered). We add a reference counter to each subscription filter. Then when a new subscription is received, pre-existent buffers are checked for identity. If none identical filter is found, a new buffer is created with its counter initialized to one. If an identical filter is found, its counter is incremented. When an unsubscription is received, its counter is decremented. Finally, when a counter reaches to cero, it is disposed.
- *Filter covering:* a filter F_1 covers another filter F_2 (denoted $F_1 \sqsupseteq F_2$) iff $N(F_1) \supseteq N(F_2)$. This function defines a reflexive partial order over the set of all possible existing filters. If $F_1 \sqsupseteq F_2$, then the fact of a notification $n \in N(F_2)$ implies that $n \in N(F_1)$. Moreover, F_1 is a *real cover* of F_2, denoted by $F_1 \sqsupset F_2$, iff $N(F_1) \supset N(F_2)$. Checking if a filter covers another allows us to benefit from the already stored notifications, while avoids looking into every pre-existent buffer. When a subscription is received and a new buffer entry must be created, its filter is first checked to see if it is covered by other indexed filters. For those who does, their contained notifications are copied into the newly allocated filter buffer.
- *Filter merging:* a filter F is a *merger* of (or covers) a set of filters $\{F_1, ..., F_n\}$, denoted $F \sqsupseteq \{F_1, ..., F_n\}$, iff $N(F) \supseteq \{\bigcup_i N(F_i)\}$. F is said to be a *perfect* merger if the equality holds, and an *imperfect* merger otherwise. With this powerful function, it is possible to add new layers of filters as the caching structures grow, which allow a quick detection of pre-existent matching filters. The filters of these new layers might be created as mergers of the subordinate filters, behaving somehow like skip-lists.

The approach presented above must still address two issues which weren't yet described in detail:

- *Notification replication:* When new notifications arrive, they are checked to see in which buffers should be enqueued (i.e., copied). This might produce several replicas of the received notifications, wasting valuable caching storage space.
- *Notification ordering:* A problem with the proposed structure is how to order different buffers' matching notifications when a new buffer is allocated. This is because there is no relative order between notifications from different buffers.

A solution to both problems results by working out the combination of the previously described ideas. We develop the structure behavior which takes advantage of the content-based filter relationships previously enumerated and addresses the notification replication and ordering issues. The proposed structure is illustrated in Figure 4.2 and its definition is depicted in Pseudocode 4.1, where the following modifications were considered:

1. We define a record, *CountedMessage* (line 1), consisting of <*notification, reference counter*> pairs. This record shapes the data type for the entries of the main `buffer` (line 6), which is a circular array with a bounded length defined (statically or dinamically) by *MaxGlobalBuffers*.
2. We define another record, *CountedFilter* (line 7), consisting of the triple <*filter, reference counter, references buffer*>. This record shapes the data type for the entries of the `index` (line 13), which is an array with a bounded length defined by *MaxIndexedFilters*.
3. *CountedFilter*'s reference counter field counts subscriptions received with a given filter. This avoids duplicate buffers for identical filters. When it reaches to zero, the index entry must be deleted.
4. In order to store each notification only once, we add an additional indirection level (*CountedFilter*'s third field at line 11). Hence, filters buffer's entries *reference* notifications stored in the main buffer instead of *storing* the notifications themselves. These queues may grow up to a maximum length defined by *MaxFilterBuffers*. Note that it is reasonable to state that $MaxGlobalBuffers > MaxIndexedFilters * MaxFilterBuffers$.

Pseudocode 4.1 Buffering Structure Data Definition

```
 1: type CountedMessage = record
 2:         begin
 3:             Message msg,
 4:             int referenceCounter
 5:         end
 6: CountedMessage buffer[MaxGlobalBuffers];
 7: type CountedFilter = record
 8:         begin
 9:             Filter f,
10:             int referenceCounter,
11:             (ref CountedMessage) messages[MaxFilterBuffers]
12:         end
13: CountedFilter index[MaxIndexedFilters];
```

The processing of new messages is described in Pseudocode 4.2. The operation *enqueueNotification* simply stores the notification in the first free entry of the buffer[1], and its reference counter is initialized to 0. For each matching filter F, the operation *enqueueReference* inserts into F's

[1]Provided that by definition *MaxGlobalBuffers* is always greater than $MaxIndexedFilters * MaxFilterBuffers$, we can ensure that there is always at least one free cell.

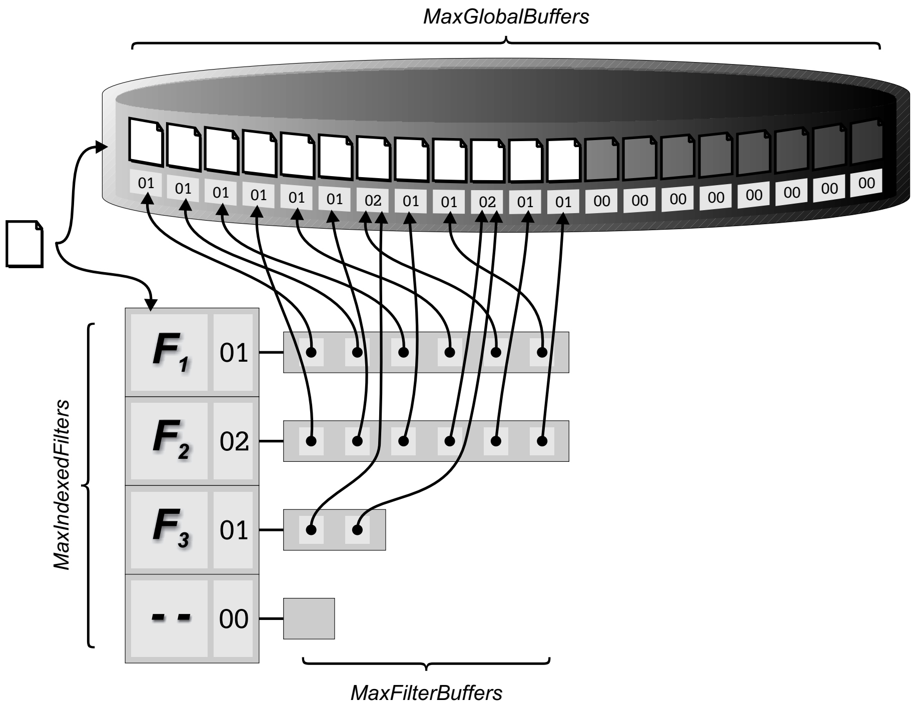


Figure 4.2: Buffering structure

associated buffer a *reference* to the notification m, incrementing `buffer[m]`'s reference counter. If the filter's buffer contains free space (constrained by *MaxFilterBuffers*), this reference cell is simply enqueued. However, if the filter's buffer is full, the first notification reference (i.e., the oldest) must be removed in order to make space for the new one. Note that reference removal includes decrementing the reference counter of the associated notification. Moreover, when the reference counter reaches to zero, the notification is removed from the main buffer. Finally, the operation *incrementReferenceCounter* is quite straightforward: it simply increments the specified reference counter value.

Pseudocode 4.2 Processing of a message m

```
1: enqueueNotification(buffer,m)
2: for all $F \in index \cdot m$ matches $F$ do
3:     enqueueReference(index[F].messages, m)
4:     incrementReferenceCounter(buffer[m])
5: end for
```

The processing of a subscription's filter F is depicted in Pseudocode 4.3. Basically there are two options: the structure might already contain an *identical* filter (from another, previous subscription) or not. If this is the case, its reference counter is incremented (line 2). Otherwise, the following process is executed: First, space for a new buffer is allocated in the index (line 4), constrained by the limit *MaxIndexedFilters*. If the structure has no space for further indices, the subscription can not be cached and a warning message is logged. Second, pre-existent filters which *cover* F are searched for. For each of their associated buffers' notifications, operations *orderedEnqueueReference* and *incrementReferenceCounter* are executed. The former operation implements ordered insertions on *filter's* buffers according to the position in the *main* buffer; the latter was described previously.

The processing of an unsubscription's filter F is depicted in Pseudocode 4.4. Initially, the reference counter at *index[F]* is decremented. When it reaches to zero, the entry (and its associated message references buffer) must be disposed. This comprises the following tasks: for each referenced notification, its reference counter is decremented (line 4); if this reference counter reaches to zero, the operation *removeNotification* is executed, which erases the notification from the structure (line 6); and finally, index entry can be deleted (line 9).

Pseudocode 4.3 Processing of a subscription's filter F

```
1: if $\exists$ $G \in$ index $\cdot$ $G$ is identical to $F$ then
2:    incrementReferenceCounter(index[G])
3: else
4:    allocate space for new buffer at index[F]
5:    for all $G \in$ index $\cdot$ $G$ covers $F$ do
6:       for all $m \in$ index[G].messages $\cdot$ $m \notin$ index[F].messages do
7:          orderedEnqueueReference(index[F].messages, m)
8:          incrementReferenceCounter(buffer[m])
9:       end for
10:   end for
11: end if
```

Pseudocode 4.4 Processing of an unsubscription's filter F

```
1: decrementReferenceCounter(index[F])
2: if index[F].referenceCounter = 0 then
3:    for all $m \in$ index[F].messages do
4:       decrementReferenceCounter(buffer[m])
5:       if buffer[m].referenceCounter = 0 then
6:          removeNotification(buffer,m)
7:       end if
8:    end for
9:    dispose entry space at index[F]
10: end if
```

As we stated previously, notification storage and cache querying are not independent of each other. As we will see next, and in spite of having a precise definition of a buffering data structure behavior, some of the described procedures might slightly change according to different caching strategies and deployment configuration (e.g., routing algorithms). We proceed by introducing these strategies and analyzing different possibilities for implementing them over different deployment configurations.

4.4 Cache Querying

Before reasoning on event-based systems cache querying, some preliminary issues must be clarified. In principle, we must note that the broker's network conform an *undirected acyclic graph (UAG)*. This UAG is depicted in Figure 4.3. Note that in the illustration, components belonging to the notification service have gray backgrounds, while clients *(X_i)* always have a white background. Each client X_i can maintain more than one active subscription at a time. Additional terminology will help us to provide a more precise explanation. We use *Local Event Broker (LB_i)* to refer to the client's access broker (a pub/sub library), which provides the pub/sub interface used by clients to produce or consume information. Complementary to this, other brokers exist which act as event routers *(ER_i)*. They can be further classified as *Border* or *Inner Brokers*, though in practice they are the same kind of components. Finally, empty *routing tables* (previously defined in section 2.5.1) are also shown.

With this graph in mind, a first issue arises: given that the notification service is an UAG, a key aspect is *how deep* should the cache query search for notifications, or, more formally, *how many levels* (lets say, k) should be traversed in the search for notifications stored at other broker's buffers. We argue that the amount of traversed nodes is a crucial aspect of the strategies, since each link traversal must be viewed essentially as a Remote Procedure Call (RPC), which results in the following costs:

- serialization/deserialization of the notification or reply
- marshalling/unmarshalling of parameters

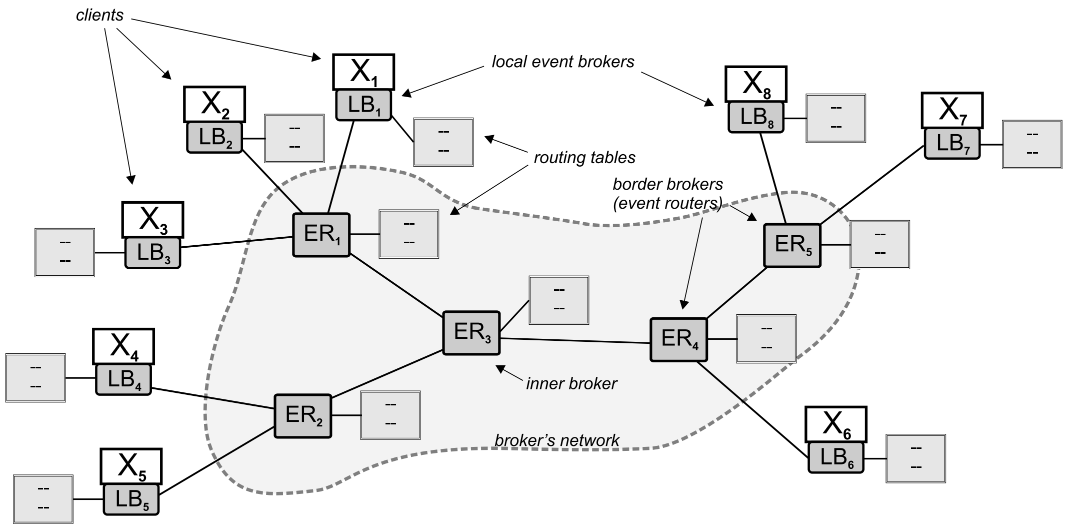


Figure 4.3: UAG components

- RPC runtime and communication software
- physical network transfer

These costs point out that a single RPC (or traversal) requires approximately between 10.000 and 15.000 machine instructions, thus having local procedure calls about 100 times faster.

In spite of the previously mentioned, it might be the case that a notification request (from a client subscription) can be satisfied by fetching notifications cached in several broker's buffers. Hence, considering that these requests always originate at the borders of the graph, we can classify the approaches according to the number of traversed levels k:

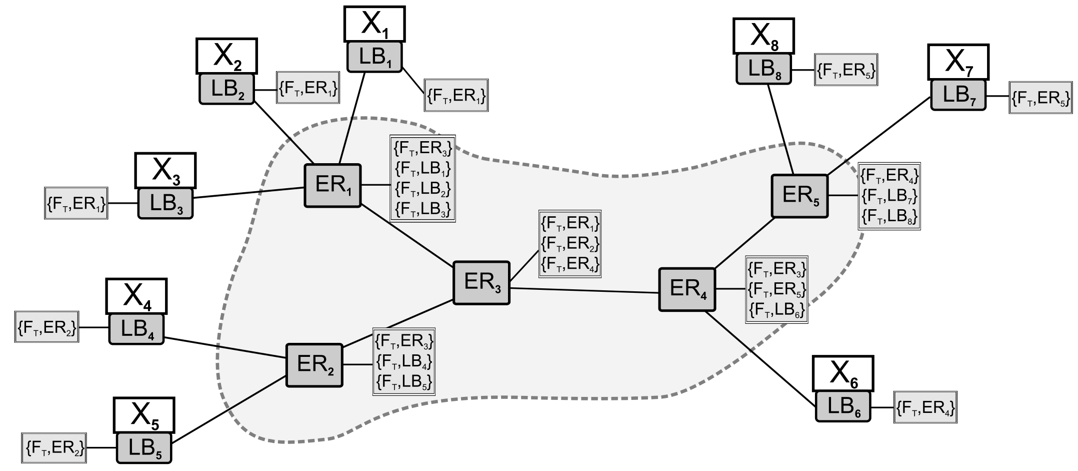
- *With $k = 1$*, the strategy only searches the caches at Local Event Brokers.
- *With $1 < k <= M$*, the strategy can further ask neighbor brokers for cached notifications, but its actions are restricted to a maximum depth *M*.
- *With $k = \infty$*, the strategy is allowed to perform as an exhaustive search.

Next, we name these strategies, pose their difficulties and develop solutions to overcome them as well.

4.4.1 Local Broker Caching

The first approach to caching in event-based systems is to deploy the described data structure at the Local Event Brokers. We can attach the caching strategy to the `RoutingEngine` in the home stub layer (described in Section 2.5.1) as illustrated in Figure 2.7. Just as with the rest of the strategies, the "glue code" that binds the broker behavior with the caching strategy is provided by an abstract component that incorporates the required functionality (e.g., intercepts the appropriate calls).

We elaborate this approach in conjunction with a deployment of the *Flooding* routing algorithm. Hence we will first illustrate the behavior of the former alone. In Figure 4.4 the initial state of the routing tables is shown. As it can be seen, each routing entry contains a filter F_T, which matches any notification, together with their couples which point to every neighbor broker. These entries allow the REBECA routing framework to flood published notifications. Then, the subscription process is depicted. In Figure 4.5.a, X_1 subscribes to F, while in 4.5.b, X_4 subscribes to G (lets assume that F and G are disjoint, or more precisely, $N(F) \cap N(G) = \emptyset$). As it can be seen, the filters are simply added to the respective Local Event Brokers' routing tables[2]. Finally, the notification forwarding process is depicted. In Figure 4.6.c, X_5 publishes n_1, which only matches F. As it can be seen, only client X_1 receives the *notify()* message even though the notification n_1 floods the whole broker network.



X1 LB1 {FT,ER1}
X2 LB2 {FT,ER1}
X3 LB3 {FT,ER1}
X4 LB4 {FT,ER2}
X5 LB5 {FT,ER2}
X6 LB6 {FT,ER4}
X7 LB7 {FT,ER5}
X8 LB8 {FT,ER5}
ER1 {FT,ER3} {FT,LB1} {FT,LB2} {FT,LB3}
ER2 {FT,ER3} {FT,LB4} {FT,LB5}
ER3 {FT,ER1} {FT,ER2} {FT,ER4}
ER4 {FT,ER3} {FT,ER5} {FT,LB6}
ER5 {FT,ER4} {FT,LB7} {FT,LB8}

Figure 4.4: Flooding behavior: initial deployment

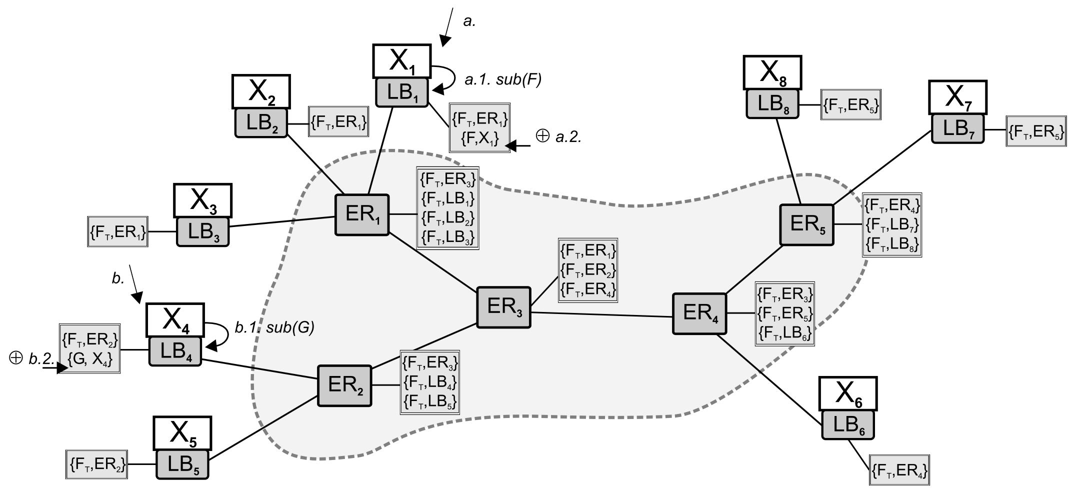


Figure 4.5: Flooding behavior: subscription update

In such an environment, the addition of the caching feature at the Local Event Brokers is illustrated in Figure 4.7. Attached to the Local Event Broker's routing engines can be seen the representations of the caching data structure instances. Note that in order to react and perform, the strategy doesn't require to cross the notification service boundary, hence entirely running on the clients' machines. This very naive approach presents a series of advantages and disadvantages, which we enumerate next[3]:

△ Particularly in mobile environments, and because of the exposed in Section 2.2, the traversal of the link from the access broker to the Border Broker might be very expensive. Hence in those deployments where Local Event Brokers act as hubs for several (mobile) client applications, this approach could perform well.

△ The notification storage behavior previously described remains unchanged (i.e., no further modifications need to be developed).

[2]In these schemes the symbols ⊕ and ⊗ represent routing entries additions and removals, respectively.
[3]In these analysis the symbols △, ▽ and ≡ represent advantages, drawbacks and relative, respectively.

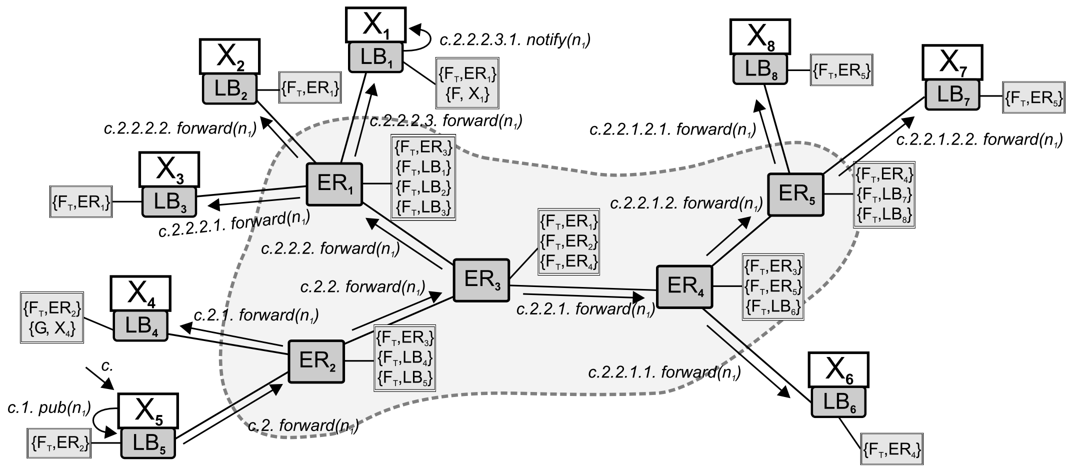


Figure 4.6: Flooding behavior: notification forwarding

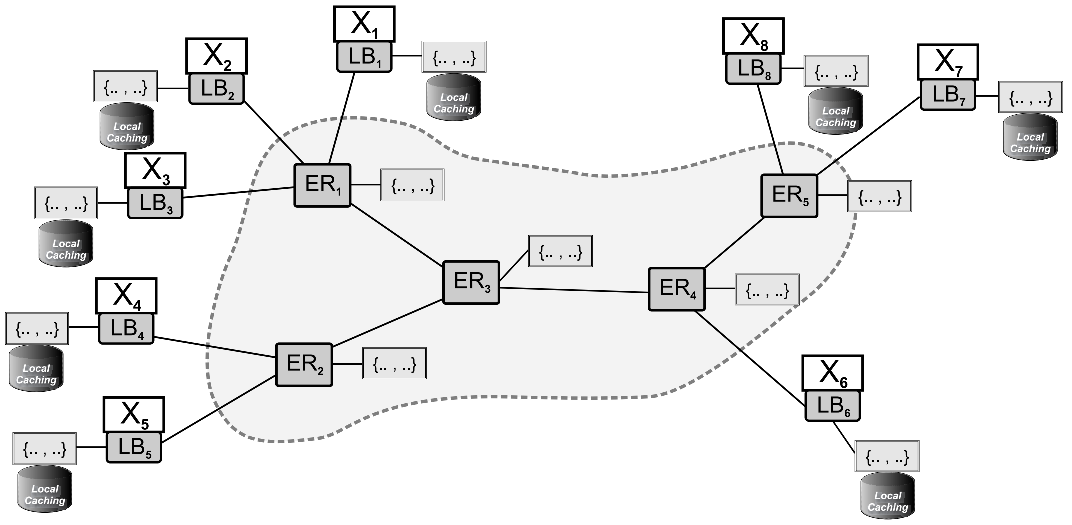


Figure 4.7: Flooding behavior with Local Broker Caching

△ No extra requirements from the infrastructure.

▽ Given that notification fetching is restricted to Local Event Broker's buffers, the effectiveness of this strategy heavily depends on the *similarity* of the interests of the clients too (more precisely, the similarity of the issued subscriptions). This is the foundation of an analysis which evaluates caching efficacy according to variation on similarity of nearby clients interests.

≡ If the routing algorithm being in use is *Flooding*, no other better alternative can be chosen.

The latter point requires further explanation. Remember from Section 2.5.2 that the remaining routing algorithms rely on filter forwarding to update the routing configuration in reaction to subscribing and unsubscribing clients. This allows the brokers to update their routing tables accordingly. The whole goal is to prevent unnecessary notification forwarding. However, and as it was shown in Figure 4.5, filters are *not* forwarded when Flooding is used. Hence, a caching structure deployed at inner or (even) Border Brokers wouldn't be able to base its decisions on observed filters (since they don't see filters at all). Furthermore, the fact of a notification being forwarded is not enough to assume that the notification itself is of interest for at least one client (it could be finally discarded by every Local Event Broker in the notification forwarding process). Thus, we conclude that implementing a strategy other than Local Event Broker Caching on top of a Flooding deployment is not reasonable because: a) every notification should be stored without knowing if they are useful for anyone, and b) filters should be forwarded into the broker's network, completely missing the point of the routing algorithm selection.

This strategy possess a k value of 1. In the forthcoming subsections we present other approaches which tackle this strategy's main drawback, which is the dependency on nearby clients similarity of interests, by augmenting k's value.

4.4.2 Border Broker Caching

In order to increase the possibility that new subscriptions find other identical or overlapping ones, the next step consists of moving the point where notifications are stored inside the broker's network.

The basic idea is to perform the notifications caching at the Border Brokers and devise an interaction with the Local Event Brokers to fetch these notifications. We could imagine *information cones* where clients embraced by each cone consume similar information, or what is the same, issue similar subscriptions. The classified scenarios in Section 3.1 further make it reasonable to state that in mobile environments nearby clients will have similar interests. The hope is that notifications cached for a client will be needed by other local clients.

By assuming that a traversal of the link between Local Event Brokers and Border Brokers can be made, the number of cached notifications for distinct subscriptions is clearly maximized. It is also assumed that *Simple Routing*, a more complex routing algorithm, is used by the brokers. This is because in order to appropriately deploy the caching structure at the Border Brokers, they must always receive the subscriptions, which only occurs with this algorithm.

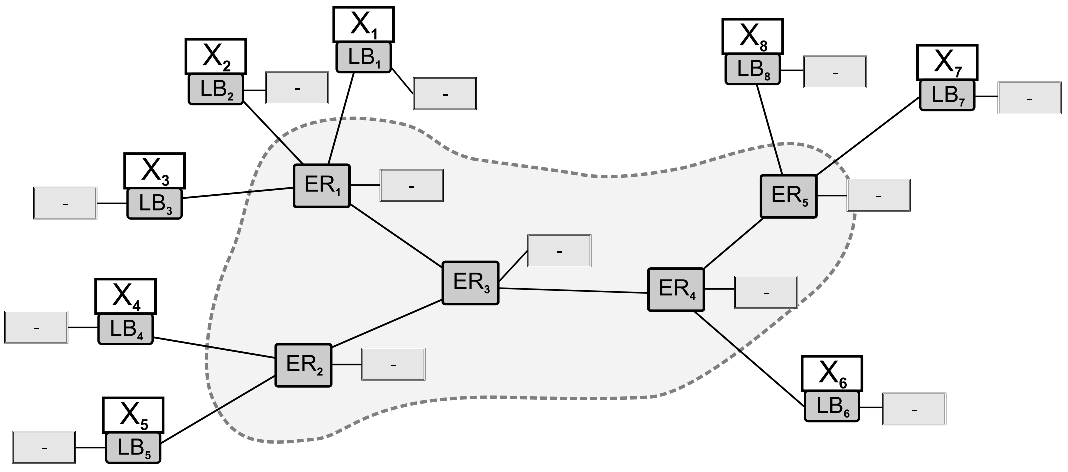


Figure 4.8: Simple Routing behavior: initial deployment

In the same way we did before, we elaborate this approach in conjunction with a deployment of the *Simple Routing* routing algorithm. Hence first we will illustrate the behavior of the former alone. In Figure 4.8, the initial state of the routing tables is shown. In contrast to that of Figure 4.4, the routing tables are empty. With *Flooding*, the routing table's entries change according to the network topology (which is rather static). In contrast, the other routing algorithms rely on filter forwarding, hence the routing tables reflect a global or partial knowledge of active subscriptions (which are rather dynamic). The subscription process is depicted next. In Figure 4.9.a, X_1 subscribes to F. As it can be seen, the filter is not only added to LB_1's routing table but *flooded* across the network. Through appropriate administrative messages, filters are added into every broker's routing table. Finally, the notification forwarding process is depicted. In Figure 4.10.b, X_5 publishes n_1, which matches F, while in 4.10.c, X_6 publishes n_2, which doesn't match F. As it can be seen, client X_1 only receives n_1. Note also that n_2 is not forwarded beyond LB_6 and that n_1 is only forwarded across a route towards client X_1.

In such an environment, the addition of the caching feature at the Border Brokers is illustrated in Figure 4.11. Note that although event router ER_3 is currently an *Inner* Broker, nothing prevents

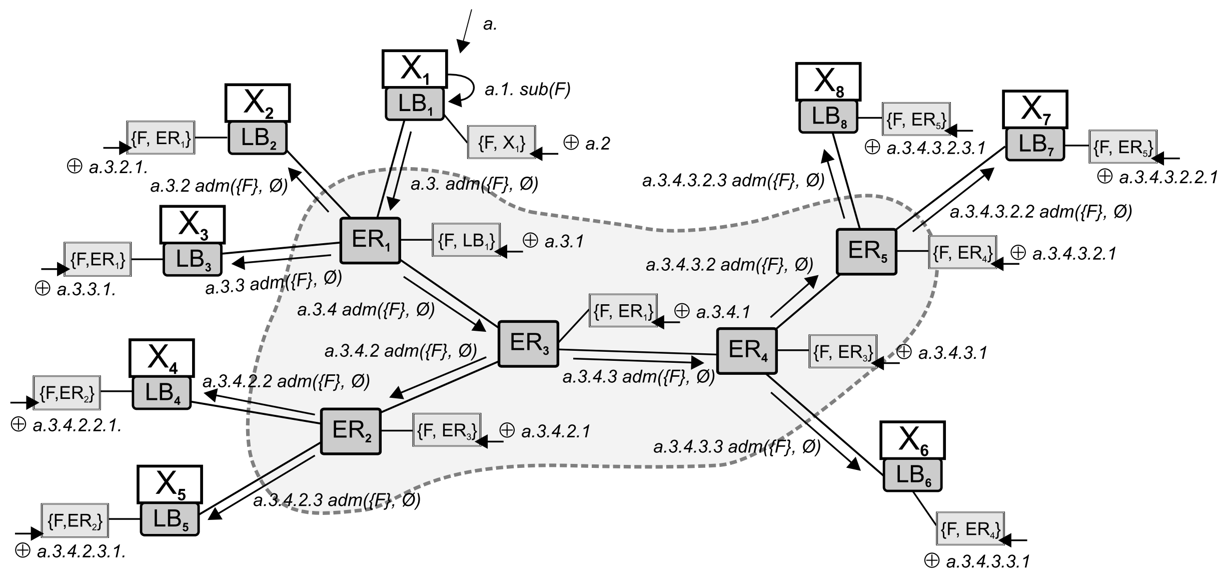


Figure 4.9: Simple Routing behavior: subscription update

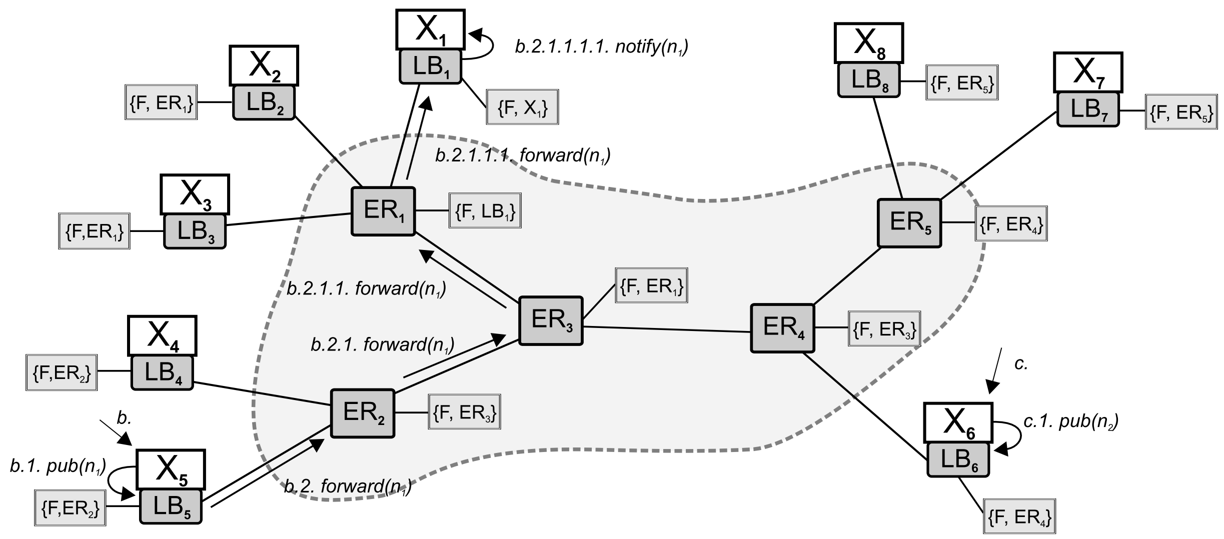


Figure 4.10: Simple Routing behavior: notification forwarding

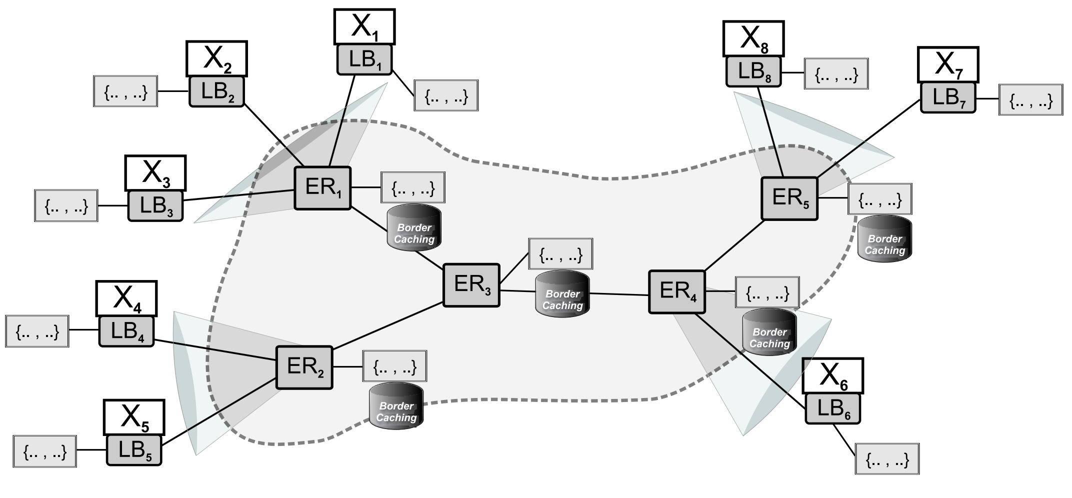


Figure 4.11: Simple Routing behavior with Border Broker Caching

it of getting converted into a *Border* Broker dynamically (e.g., by receiving a connection from new clients). Hence, it has the caching structure attached too.

The interaction is rather simple: the Local Event Broker asks the Border Broker for notifications matching a given filter and bound, and the former, in turn, answers with a reply containing the required notifications, if any. This interaction, though, must hold the requirement that when a client issues a subscription, the infrastructure must start delivering back notifications (from the past or not) *in the same order* that would have done if this client had subscribed earlier. More precisely, duplicate or missing notifications must be prevented. Thus, a couple of considerations should be made.

The first consideration is to tie the notification fetching functionality to the subscription forwarding process in an *atomic* operation. This allows the routing framework to immediately start routing notifications to the client after the reply is delivered back. The second consideration is based in that *reply's* notifications and *new* notifications are concurrent, hence careful attention must be paid to what to do with these concurrent notifications to deliver them in the correct order. With this purpose, the Local Event Broker blocks those notifications that should be delivered to the client which were received in the meantime between the reply is requested and it is effectively received. Together with each blocked notification, the link from where it came from is registered. Afterwards, when the reply arrives from link ℓ, the amount of blocked notifications that came from link ℓ are compared to the amount of notifications contained in the reply. If the former are more, the latter are *dropped*, and viceversa (otherwise duplicates could be delivered). Finally, the chosen notifications are delivered.

Hence, the subscription process is slightly changed. When a client issues a subscription with a request for notifications from the past, the strategy adds the bound information to the subscription. When a Local Event Broker receives a subscription which includes bound information, it inserts a marked routing entry in the routing table. This marked entry indicates that the entry is set on *hold*. Moreover, an initially empty *reply* is associated to the entry. This reply will block and retain the delivery of matching notifications (along with the link where they came from) received in the meantime while the Local Event Broker receives a reply from the Border Broker. Figure 4.12 illustrates the main flow of events in an UML sequence diagram:

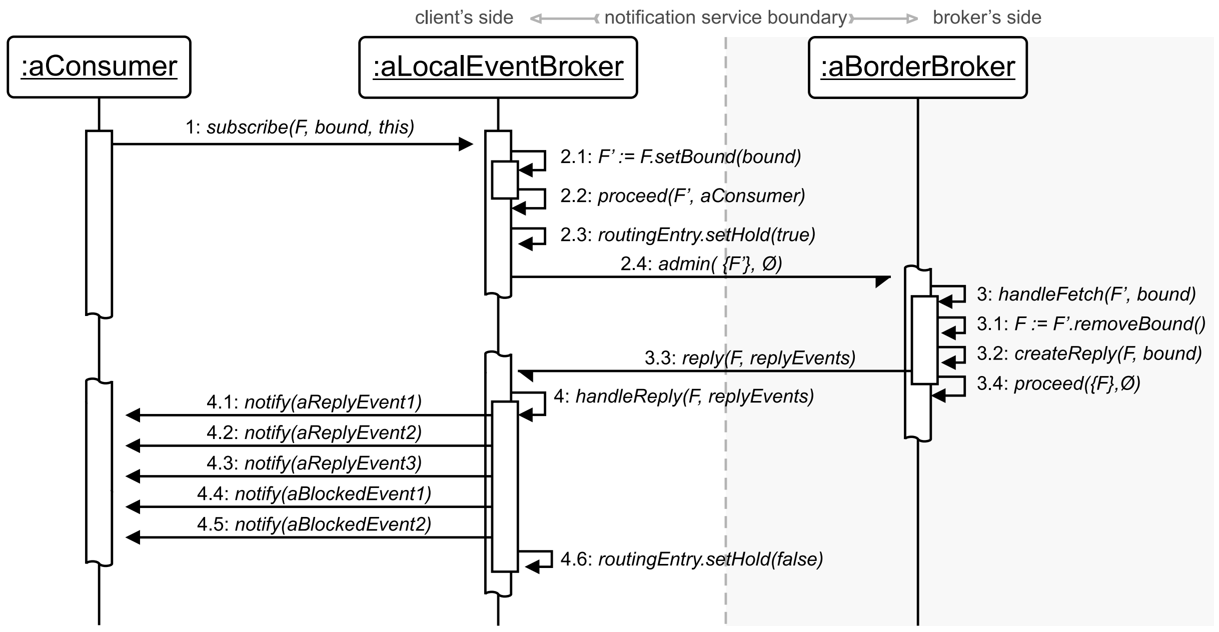

client's side
notification service boundary
broker's side
:aConsumer
:aLocalEventBroker
:aBorderBroker
1: subscribe(F, bound, this)
2.1: F' := F.setBound(bound)
2.2: proceed(F', aConsumer)
2.3: routingEntry.setHold(true)
2.4: admin({F'}, Ø)
3: handleFetch(F', bound)
3.1: F := F'.removeBound()
3.2: createReply(F, bound)
3.3: reply(F, replyEvents)
3.4: proceed({F},Ø)
4: handleReply(F, replyEvents)
4.1: notify(aReplyEvent1)
4.2: notify(aReplyEvent2)
4.3: notify(aReplyEvent3)
4.4: notify(aBlockedEvent1)
4.5: notify(aBlockedEvent2)
4.6: routingEntry.setHold(false)

Figure 4.12: Border Broker Cache Querying

1. A consumer subscribes itself to a filter and specifies a bound (F and `bound`, respectively).
2. The original Local Event Broker behavior is preceded by a new action:

2.1 First, the `bound` information is attached to the subscription *F*, which we denote by *F'*. Hence this specialized `Filter` also carries information about the specified bound (e.g., *how many* notifications are required).

2.2 The subscription process proceeds as usual, except that now *F'* is forwarded (instead of *F*) to the Border Broker.

2.3 The usual behavior creates a routing entry with the subscription *F'*. Since the subscription carries `bound` information, the entry is set on hold. Also, a new (empty) reply retains meantime received events.

2.4 With *Simple Routing*, every subscription is forwarded through appropriate asynchronous `admin` messages (note different arrowheads).

3. The Border Broker at the other side receives the `admin` message including a bounded filter, which executes the notification fetching functionality:

   3.1 First the `bound` information is extracted from the subscription, converting *F'* again in *F*.

   3.2 The caching data structure processes the subscription (as indicated in Pseudocode 4.3 in page 27). This involves the allocation of an index entry (if none identical is found) and the retrieval of existing matching notifications. With these notifications (if any, and considering the specified bound) a reply is created.

   3.3 This newly created reply is delivered back to the Local Event Broker as an special administrative `reply` event.

   3.4 The subscription forwarding process proceeds as usual for the routing algorithm, except that now *F* is forwarded instead of *F'*. This might involve the forwarding of *F* to other neighbor event brokers.

4. At some point in time, the Local Event Broker receives from link ℓ the administrative event identified as a `reply`, which fires its execution handling:

   4.1 - 4.3 Each of the events contained in the reply are unpacked and delivered back to the client through the traditional `notify()` callback method.

   4.4, 4.5 Blocked events received in the meantime by the Local Event Broker which didn't come from the link ℓ and that were locally stored in the associated queue are afterwards delivered, if any.

   4.6 Next, the routing entry associated to *F* is released from the *held* state, cleaning up its associated notification queue.

The only potential complication with this approach would be that notifications received in the meantime at the Local Event Broker already fulfill the request. If this alternative to the main flow of events occurs, waiting for the reply is no longer necessary because the notifications included in it would be too old. Hence, the locally queued notifications are automatically delivered and the queue is disposed. With this modification, the bootstrapping delay of a consumer is guaranteed to be no larger than without recent history in the worst and better in the average case. On the other hand, in order to provide a fault-tolerant strategy, it would be clever to also set a *timer* when a subscription is set on hold. In case the Border Broker is too slow or suddenly down, it is preferable to let the routing algorithm to effectively deliver queued notifications instead of keep blocking them, perhaps forever. In both cases (local queue fulfilled request or time-out received), if the reply eventually arrives later, it can simply be discarded.

In an analogous way, the unsubscription forwarding process indicates to the caching structure that the buffered entries are no longer needed and can be disposed. This strategy is a little bit more complicated than the previous one. Though, the biggest problem which is mixing new notifications with the reply is quite straightforward to solve. The reply is prepared only at the Border Broker so new notifications cannot overtake the reply. Next we enumerate a brief advantages and drawbacks list:

△ In comparison with the previous approach, the quality of service is enhanced because each cone embraces more distinct subscriptions than each Local Event Broker can.

△ The workload is delegated to the middleware: the *processing* is mainly performed at the infrastructure itself (in contrast with the home stub layer), while the notification *storage* is completely done at the Border Brokers.

△ The whole interaction protocol is built upon the existing links. It is not necessary to create new sockets or communication channels. Only a new `Reply` administrative message must be recognized.

▽ Although this strategy doesn't have requirements from the infrastructure, filter forwarding routing algorithms like *Simple Routing* or *Covering-based Routing* assume that it is possible to uniquely identify each filter, restricting the possible deployment configurations it can be used with.

▽ When a client issues a subscription and its Border Broker doesn't have enough past notifications to fulfill the request but a neighbor does, this strategy can't do nothing about it (i.e., there are no cooperative actions between neighbor event brokers).

Though with a k value of 2 the *QoS* is enhanced, the restriction to search only for notifications at the Border Broker needs to be suppressed for practical relevance. Next we develop another strategy where k is extended up to an arbitrary number.

4.4.3 Merging Caching

The last drawback mentioned in the previous approach leads to another strategy. In a real environment it is reasonably to think that each Event Router will be deployed on machines with different processing and storage capabilities. This variability in the Event Routers' storage capabilities can be translated into different caching structures sizes, or, more precisely, different values for *MaxIndexedFilters* and *MaxFilterBuffers*. Be that as it may, it could happen that the desired number of past notifications may not be available at the Border Broker.

Furthermore, in a more complex scenario, several clients might concurrently produce information which is consumed by other clients. For example, in Figure 4.13 we extend the scenario of Figure 4.10 where both producers X_5 and X_6 are publishing notifications that match the client X_1's subscription's filter F. Initially, client X_1 has issued a subscription with a filter F, and given that *Simple Routing* is being used, F was spread across the broker's network.

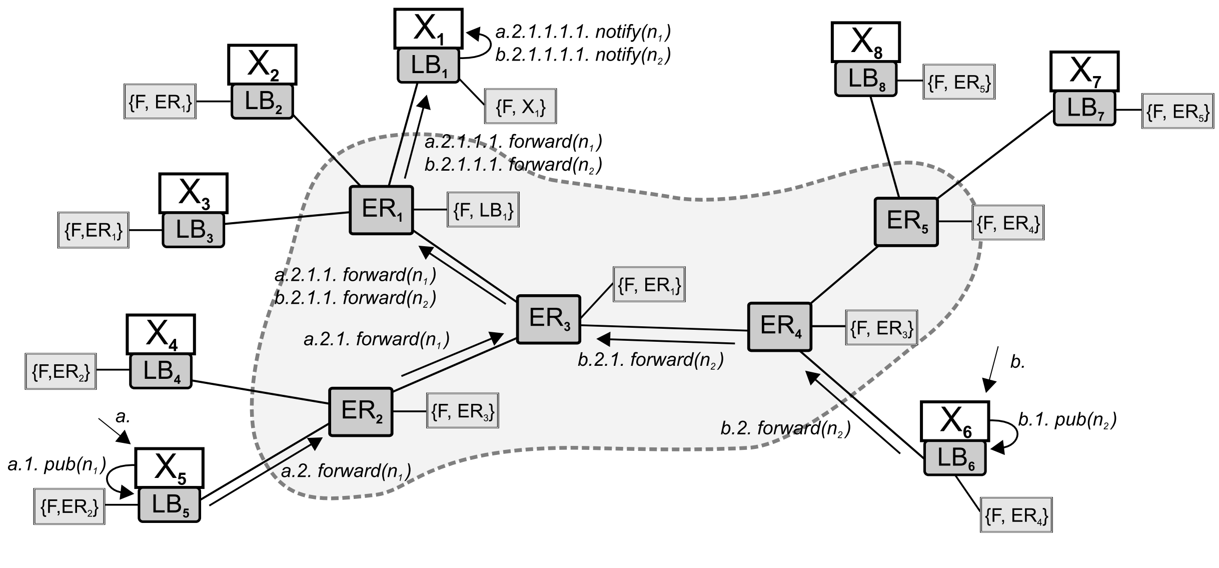


Figure 4.13: Multiple producers scenario

Now assume that client X_3 issues a subscription with a filter G (which is covered by F), and a bound of 50 notifications. With the previous approach, the best effort that can be achieved is to fetch and deliver the notifications from Border Broker ER_1. Client ER_1 could have only 25 notifications stored (e.g., ER_1's *MaxFilterBuffers* value is 25), though. Nevertheless, stored somewhere else upstream in the broker's network, we could find other matching notifications, which would have been delivered if the client had effectively subscribed there earlier. This is the case for broker ER_3 if its *MaxFilterBuffers* value is greater than 50, for instance.

Even when every broker has the same *MaxFilterBuffers* value the strategy performs better, since as the query gets closer to individual producers, the notifications stored in each broker's cache will belong to less producers. For example, client X_2's new subscription will result in two new delivery paths: $< LB_2, ER_1, ER_3, ER_2, LB_5 >$ and $< LB_2, ER_1, ER_3, ER_4, LB_6 >$. For these paths, the Event Router ER_3 is also called a *junction broker*, because it is the junction point for several delivery paths. On one hand, brokers ER_1 and ER_3's caches store notifications both from producers X_5 and X_6. On the other hand, broker ER_2's cache only stores notifications from X_5, while broker ER_4's cache only stores notifications from X_6. The basic idea of this approach is to extend the number of queried brokers up to the point where the query reaches individual producers. Instead of considering only notifications stored at the Border Broker, the former could decide to further ask its neighbor Event Routers for more notifications which could be stored somewhere else upstream in the network.

The addition of the caching feature at the Event Routers is illustrated in Figure 4.14. Attached to the Event Routers' routing engines can be seen the representations of the caching data structure instances (note their different sizes).

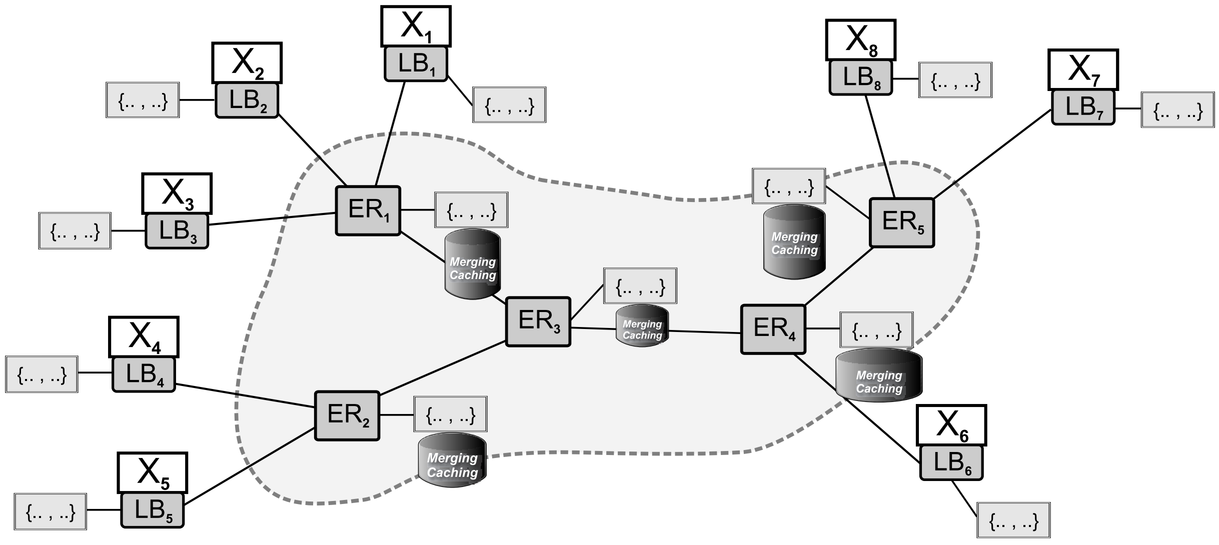


Figure 4.14: Merging Caching deployment

Some difficulties arise in supporting this caching behavior. First, upon a replay request, an Event Router ER_x (be it a Border Broker or an Inner Broker) will check if it can fulfill the request with its own stored notifications. If this is the case, the reply is created and delivered back to the requester. If not, the broker should query all peers (lets say, ER_y) in order to get a better estimation of past notifications. Hence a mechanism must be devised that allows to manage which peer brokers have been queried and which of them have already delivered its reply. A solution to this problem is the following: when the broker's caching structure doesn't have enough notifications stored, an empty reply is created on hold, and an empty list is attached to the reply. This list will contain the brokers that the reply is waiting for. Each queried broker is inserted in this list. When a reply is received from broker ER_y, it is removed from the reply's brokers waiting list and its notifications are merged with those of the held reply. Finally, when all replies have been received (which is recognized because of an empty waiting list), broker ER_x can deliver the reply back to the original requestor. This overall reply contains notifications already stored in ER_x's caching structure, merged with those received in the peer replies.

The second difficulty is that, in general, a specific ordering cannot be assumed between replies from different peers. Reply processing may be based on first-come-first-serve, random interleaving, or timestamp ordering if logical or global-time clocks are presupposed. However, only replies from ER_x's own caching structure will have the order of notifications the consumer would have observed if it had subscribed earlier. Next, we propose a scheme for this issue that doesn't imposes any requirements.

The notification processing described in Pseudocode 4.2 in page 26 is modified: each incoming notification is registered along with the broker's link from where it came from. Thus, it is possible to ask a broker's caching structure for stored notifications *which didn't come from a specified link.* With this new functionality, the broker querying can be extended to an arbitrary deep level (i.e., k value).

We illustrate this behavior with the previous example. Client X_3 issues the subscription for filter G and a bound of 50 notifications, so just as with the previous approach LB_3 forwards a `BoundedFilter` containing G and the bound. The caching structure of the Border Broker ER_1 might have notifications stored which came from LB_1, LB_2, LB_3 and/or ER_3. If they aren't enough to fulfill the requested bound, ER_1 decides to ask its neighbor brokers for more notifications. Since the ordering of notifications relative to the source is fixed, replies will include the same most recent notifications which allows to *drop* as many notifications from a reply as already present in ER_1's caching structure for this source. Otherwise, duplicates might be delivered.

In turn, if notifications stored in ER_3 that didn't came from ER_1 are not enough to fulfill the requested bound, ER_3 decides to ask its neighbor Event Routers for more notifications (in this case, it can ask ER_2 and ER_4), and so on. Note that at Event Router ER_2 and ER_4 the strategy will find only notifications published by producer X_5 and X_6, respectively, which was the goal of this strategy. As an Event Router receives the requested replies, it goes merging them by *dropping* as many notifications as already present in its caching structure for the reply's source.

Timers can also be used with this approach. Though, the difference lies in that as the strategy searches deeper in the broker's network, the timers must be set up with shorter periods. Thus, the bound could also carry information about the k value so far, which is used to set the timers for requests at level k. Another scheme could be to let the client specify the timer's value and recursively divide this value into fractions when a broker decides to ask its neighbors for further notifications.

Just as with the previous approach, the Local Event Broker blocks those notifications that should be delivered to the client which were received in the meantime between the reply is requested and it is effectively received. When the reply eventually arrives from link ℓ, the amount of blocked notifications that came from link ℓ is compared to the reply size: if the former are more, the latter are dropped and viceversa. On the other hand, if the retained notifications already fulfill the request or the timer expires, they are automatically delivered. The subscription is unset from the *held* state and the reply is disposed.

Next, we enumerate the main advantages and drawbacks of this approach:

- $\triangle$ By extending k's value up to an arbitrary number, the number of considered notifications is enlarged, making the strategy completely independent from the nearby clients' similarity of interests.

- $\triangle$ The strategy takes advantage of variable storage capabilities of the event routers, which is more reasonable in a real environment.

- $\triangledown$ Brute force is used when locally stored notifications don't fulfill the requested bound, hence unnecessary requests might be issued to subnets where no producers exist.

As with the previous approaches, the mentioned drawbacks leads to other optimizations. In the next subsections we develop further extensions that are not exceptions: first, we show how to support other (more complex) routing algorithms; and then, we illustrate how to avoid unnecessary requests when Advertisements are used.

4.4.4 Caching with more complex Routing Algorithms

Up to this point, the caching strategies were shown in conjunction with *Simple Routing*. With this algorithm, new and cancelled subscriptions are simply flooded into the broker's network such that they reach every broker. Therefore, the buffer fetching interaction between brokers was tied to the subscription forwarding process, performing both operations atomically in order to prevent missing notifications.

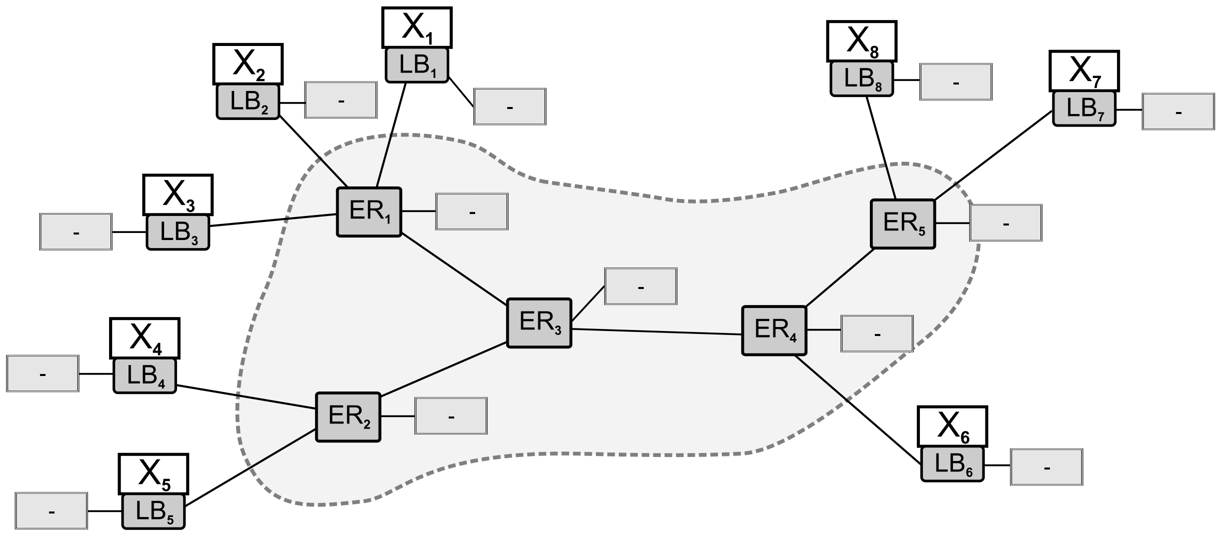


Figure 4.15: Covering-based Routing: initial deployment

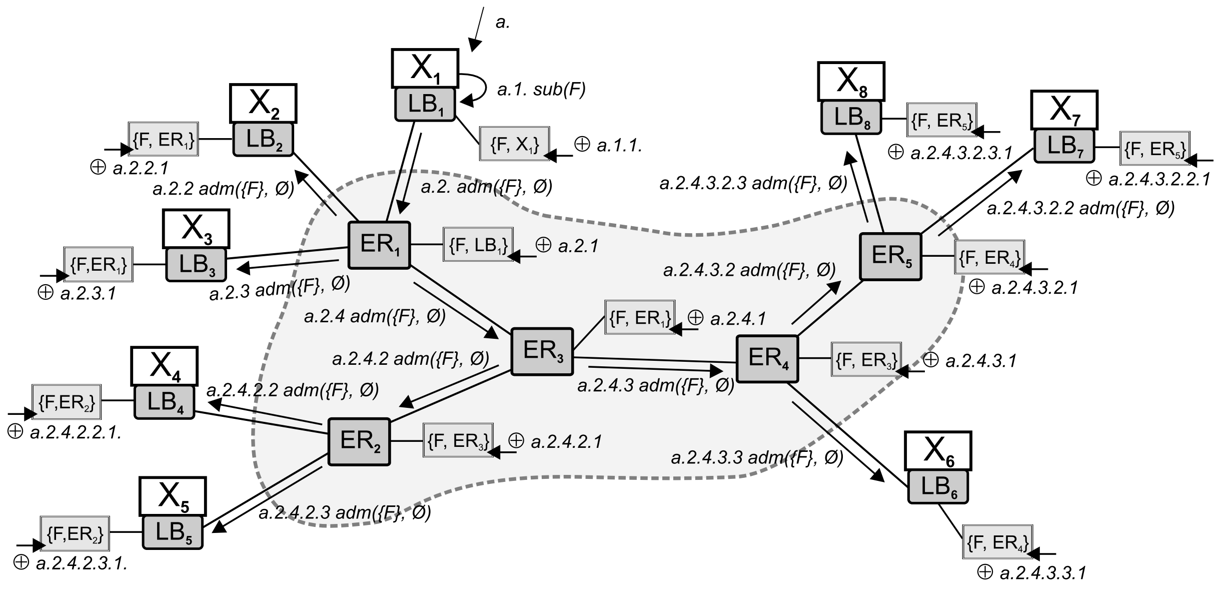


Figure 4.16: Covering-based Routing: subscription forwarding (1)

Nevertheless, when other routing algorithms are being used such as *Identity-based Routing* or *Merging-based Routing*, the subscription forwarding process might stop at some point in the broker's network. Each broker intelligently selects which brokers to forward a filter. We will illustrate the behavior of one of such algorithms: *Covering-based Routing*. In Figure 4.15, the initial state of the routing tables is shown, which is similar to that of Figure 4.8 for *Simple Routing*. The subscription process is depicted next. In Figure 4.16.a, X_1 subscribes to F. The filter is *flooded* across the network, also like *Simple Routing*. The optimization arises for the subsequent subscriptions, though. In Figure 4.17.b, X_2 subscribes to filter G, which is covered by filter F (i.e., $N(F) \supseteq N(G)$). As

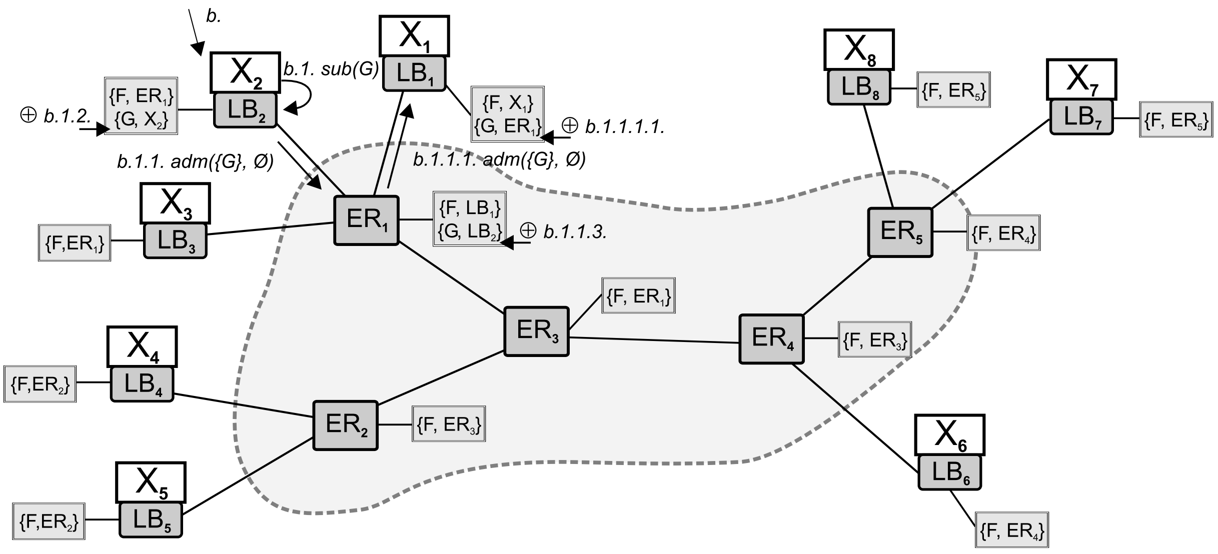


Figure 4.17: Covering-based Routing: subscription forwarding (2)

it can be seen, ER_1 did not propagate G to LB_3 nor ER_3: it knows that they already will forward notifications in ER_1's direction because of another (broader) filter, F, previously forwarded. This avoids unnecessarily forwarding the filter to other brokers, which neatly reduces the network traffic and the routing table sizes.

The problem with these routing algorithms is how to make an Event Router ER_x request a reply from broker ER_y when the routing algorithm has decided not to forward the subscription (which would include the tied request). For example, if client X_6 is a producer of notifications matching filter G and has been publishing notifications prior X_2's subscription, these notifications will be stored in all brokers at the delivery path $< LB_1, ER_1, ER_3, ER_4, LB_6 >$. Later, when client X_2 issues its subscription for filter G, the filter forwarding process doesn't go beyond ER_1. Not enough notifications could be cached there, though.

A simple solution to this problem is to create a special administrative message, `Fetch`, which includes the filter as well as the bound. This message is handled in the same way a `BoundedFilter` is, but only regarding the caching actions, that is to say, the received subscription in a `Fetch` message is not registered in the routing table. Next we develop how to extend the brokers interaction in order to forward these `Fetch` messages.

On one hand, at the requestor's side we could identify two sets of destinations links:

- L_{ra}, the set of destinies selected by the routing algorithm, and
- L_{cs}, the set of destinies selected by the caching strategy[4].

Just as with *Simple Routing*, a `BoundedFilter` can be forwarded to those brokers whose link is in L_{ra}. In contrast, to those brokers whose link is in L_{cs} - L_{ra}, the `Fetch` message is forwarded. This ensures that a reply request will be sent to every neighbor broker, even to those which weren't selected by the routing algorithm.

On the other hand, at the replier's side, the caching structure will have to create replies because of two kind of arriving events:

1. Administrative `Fetch` events which explicitly express the request for a reply.
2. Administrative subscription forwarding events which implicitly include a tied request for a reply (this is checked by inspecting the forwarded filters for a `PastBound`).

[4]By now, L_{cs} = {all but the link where the original request came from}, but in the next subsection we show how L_{cs} can be shrinked when Advertisements are used.

With this extension, the proposed caching strategies can be used with the other, more complex, routing algorithms. Now, though, the main disadvantage of extending the caching strategy to support other routing algorithms becomes clear: the goal of these algorithms, which is to avoid forwarding and storing the filter in some of the neighbors, is lost. By forcing the broker to forward the filter in spite of the routing strategy's decision, not only the network traffic is degraded but the storage is, because although the neighbor's routing table will not store a routing entry for it, the caching structure will. This tradeoff can not be avoided since the caching structure needs to know what to reply to (i.e. notifications matching which *filter*), as well as under which index entry to store the notifications.

Next, we show how to shrink the set L_{cs} in order to avoid unnecessary requests.

4.4.5 Caching with Advertisements

Advertisements are filters that are issued by clients to indicate their intention to publish certain kinds of notifications and only notifications matching an active advertisement of the respective producer should be delivered to interested consumers.

In the context of content-based routing, advertisements can be used as an additional mechanism for further optimization because it is sufficient to forward a subscription only into those subnets where matching events can be produced, i.e., where a client has issued an advertisement that overlaps with the given subscription. If advertisements are utilized for optimized routing, each broker manages two routing tables, the known subscription-based routing table and an additional advertisement-based routing table.

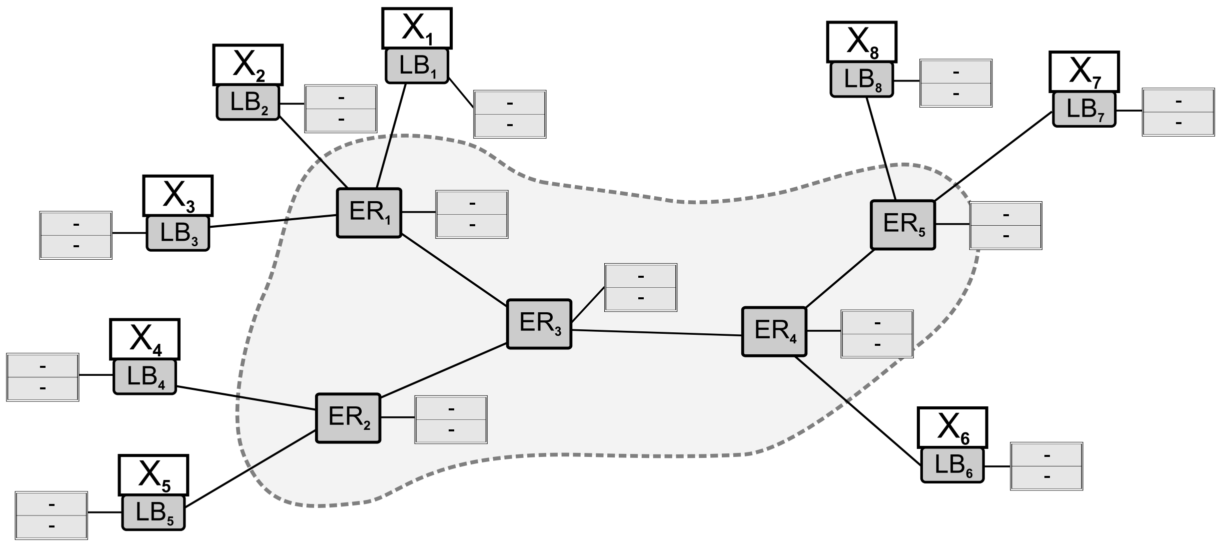


Figure 4.18: Simple Routing with Advertisements: initial deployment

As it was previously said in Section 2.5.2, a second routing table is managed by every broker. We illustrate the usage of *Simple Routing with Advertisements* in Figure 4.18. While the subscription routing tables (on the top part) are used to route notifications from producers to consumers, the advertisement routing tables (on the bottom) are used to route subscriptions from consumers to producers: a subscription is only forwarded to a neighbor if it overlaps with an active advertisement that has been received from this neighbor before. Formally, a filter F_1 overlaps another filter F_2 (denoted $F_1 \sqcap F_2$) iff $N(F_1) \cap N(F_2) \neq \emptyset$.

This advertisement routing table is maintained by the same algorithms as the subscription routing table, i.e., by forwarding new and cancelled advertisements through the broker network. For example, in Figure 4.19.a, client X_6 advertises filter F. The advertisement is flooded across the network through appropriate administrative messages and is registered in the respective tables.

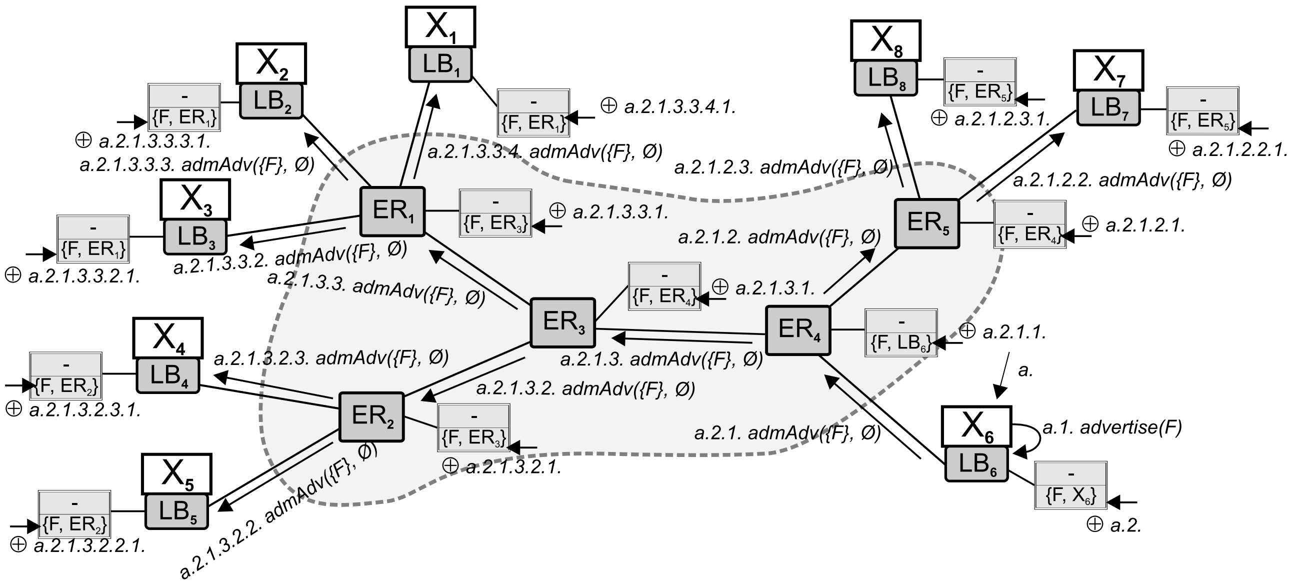


Figure 4.19: Simple Routing with Advertisements: advertisement forwarding

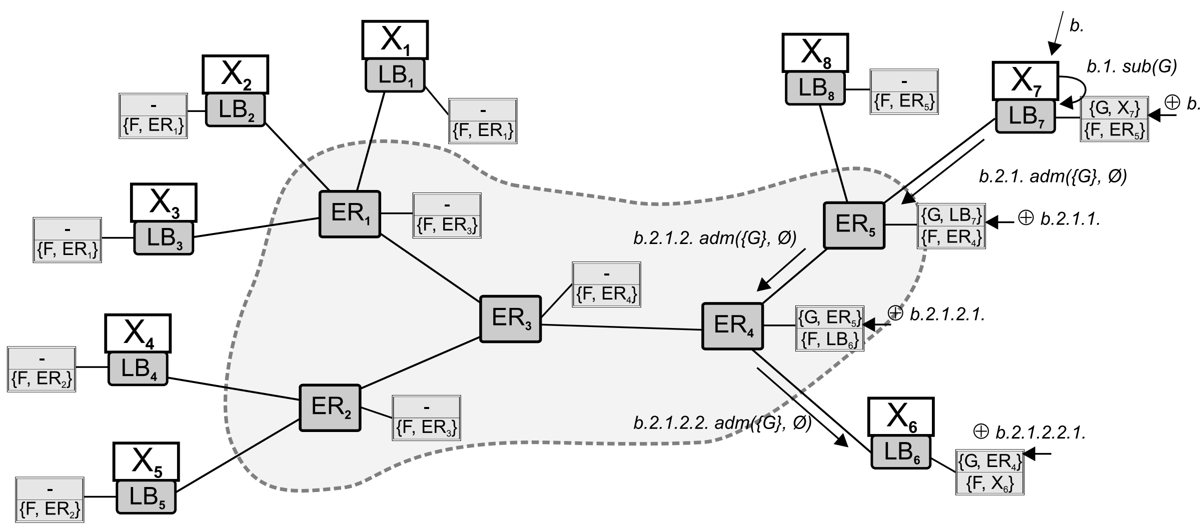


Figure 4.20: Simple Routing with Advertisements: subscription forwarding

The optimization arises when a client issues a subscription. In Figure 4.20.b, client X_7 expresses its interests for filter G, which overlaps with F. As it can be seen, in spite of using *Simple Routing*, the subscription is only forwarded towards X_6, which is the only client who can produce matching notifications.

All of the exposed above directs us to an obvious caching optimization. The information contained in the advertisements table explicitly indicates where to find producers for a given filter (i.e. through which links). Hence, when an Event Router doesn't have enough notifications stored, it can avoid asking every peer but to selectively check which links point to potential producers. This neat optimization is easy to incorporate to the previously described approach. First we redefine set L_{ra} and L_{cs} for a request for filter F:

- L_{ra}, the set of destinies selected by the routing algorithm, and
- L_{cs}, the set of destinies for which there exists an advertisement entry $<H,\ ER_y>$, where H overlaps F.

Now we analyze what to do with sources at each one:

- The links in L_{ra}-L_{cs} will be used to forward the filter as normal.
- The links in L_{cs}-L_{ra} will be used to forward the `fetch()` message.
- The links in $L_{ra} \cap L_{cs}$ will be used to forward the `BoundedFilter` (i.e., tied both the filter and the request).

With this optimization, each broker receives the appropriate message even when Advertisements are used.

4.4.6 Considering the Time Dimension

Issued subscriptions can be also specified with a time bound. In this case, the subscription asks for notifications that have been published m minutes in the past (relative to subscription time). A first approximation could be achieved by synchronizing the clocks of all border brokers. In this way, at publishing-time Border Brokers attach a timestamp to notifications in order to represent the time when they have entered into the pub/sub system. At subscription-time the border broker sets the time bound by simply subtracting the relative bound of the subscription from its local, synchronized time. This time reference is then used by the algorithm to search in the broker network for matching notifications with a newer timestamp.

Moreover, a combination of number of notifications and time into the past could also be useful, i.e., the last ten notifications within the last five minutes. This combination constrains the search in the buffers of the broker network. That means that there are two criteria to stop the search: (a) once the number of notifications within the reference achieves the solicited number, or (b) once a timestamp older than the reference is found.

4.5 Summary

In this chapter, several caching strategies have been developed that differ in several aspects like simplicity, effectiveness of notification fetching, memory overhead, etc. As it can be foreseen, other strategies could continuously emerge as a result of the planned or observed usage of the notification service.

On one hand, as more requirements from the infrastructure can be enforced, further optimizations can be made. For example, by providing *globally unique broker IDs*, each notification could be enclosed in an envelope with the ID of the producer's border broker along with a *sequence number*. This information can be used to shift the task to detect and eliminate duplicates at the event brokers. Such sequence numbers can be leveraged to eliminate duplicate sending. Requests for replies would include the sequence number of the oldest notification from a given producer still contained in the broker caching structure. This would avoid sending notifications that are known in advance that will be dropped.

On the other hand, it was shown that the applicable caching strategies depend on the deployment configuration of the notification service. The idea is to allow the network administrator to configure the notification service according to the deployed application's requirements. An initial guideline for this is resumed in Figure 4.21. The graphic is organized in three axis with several alternatives for each one: Routing Algorithms (Flooding, Simple Routing, Identity-based Routing, Covering-based Routing and Merging-based Routing), Caching Strategies (Local Broker Caching, Border Broker Caching and Merging Caching) and Usage of Advertisements (With / Without), respectively. The graphic presents two planes: each plane illustrates the allowed combinations of Routing Algorithm / Caching Strategy when Advertisements are used (rear plane) or not (frontal plane). These alternatives are resumed next:

1. As we said initially, when the routing algorithm in use is *Flooding*, the only chance is to use *Local Broker Caching*. Moreover, Advertisements cannot be used in combination to this algorithm.

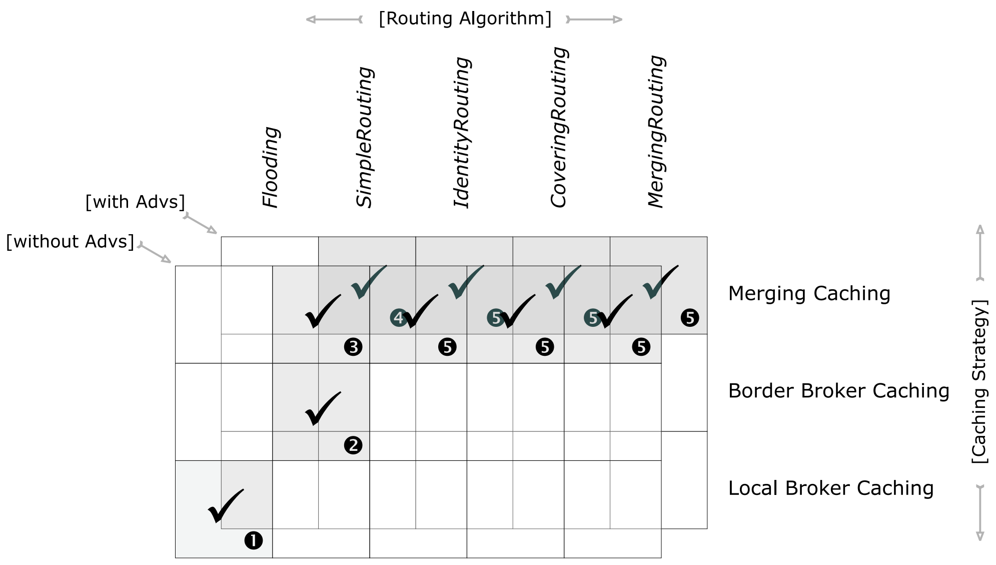


Figure 4.21: Notification Service possible configurations

2. A more effective strategy like *Border Broker Caching* can be used in conjunction with *Simple Routing*.

3. When several producers exist in the network and the event brokers storage capabilities are variable, *Merging Caching* can be applied. In conjunction with *Simple Routing*, every request for a reply is tied to the filter forwarding process.

4. If Advertisements are used, the caching strategy can take advantage of the information already present in the advertisements table to further select where to request a reply and where not.

5. Routing algorithms more complex than *Simple Routing* can also be utilized in conjunction with *Merging Caching*. Nevertheless, as we said previously, there is a tradeoff between these routing algorithms and the caching strategy since both network traffic and storage space will not be optimized.

In the next chapter we proceed with the design and implementation phases of the caching concern. As we will see, the task is divided in two steps: the architectural means that provide a framework for different strategies to run, and the caching strategies themselves that are mappings from the ideas arrived in this chapter.

Chapter 5

Caching Strategies: Design and Implementation

This chapter presents the design and implementation's decisions taken in order to develop the caching strategies. We begin by analyzing two architectural approaches. Next, we chose one and show how to progressively move from the architecture to the implementation with the help of different diagrams that guide the solution. Finally, we close the chapter with an analysis of the developed solution.

5.1 Architectural approach

The first design task was finding architectural means in order to incorporate the caching strategies into the REBECA notification service. After understanding that these strategies are heavily related to the routing algorithms, and after reading, running, testing and working with the source code responsible of the routing algorithms (Figure 5.1), we looked for a clean and adequate way to add this responsibility to the notification service.

By observing the common interface necessary for the caching strategies, we detected that basically a mechanism to intercept the messages sent to and received from the `RoutingEngine` objects was needed. Hence, two approaches came out which are described and compared below.

5.1.1 The OOP way

The first idea consists of adding a new responsibility to the `RoutingEngine`. One way to add responsibilities is by applying inheritance. Inheriting a `CachingEngine` from the `RoutingEngine` class puts the caching responsibility to every subclass instance. This approach is simple but inflexible at least for two reasons. First, the choice of caching is made statically: the objects which use the `CachingEngine` (particularly an `EventBroker` or an `EventRouter`) can't control how and when to add the responsibility to a `RoutingEngine`. Second, and more important, the `RoutingEngine` class already has its own hierarchy of subclasses that represent different algorithms for the routing concern. Thus, inheriting a `CachingEngine` class for every `RoutingEngine` subclass, trying to cover every combination of responsibilities statically, would yield an undesired class explosion.

Nevertheless, a more flexible approach was explored. The solution was enclosing the `Routing Engine` component in another object that adds the caching. A generic solution for this kind of problem was described in [18], who call the enclosing object a *decorator*. This *design pattern* is designed to add responsibilities to objects without subclassing. In our problem, the decorator component was called a `CachingEngine`, and a superclass of both classes was generalized as an `Engine`, who defines the interface that clients expect to use (Figure 5.2). This `CachingEngine` conforms to the interface of the component it decorates (i.e. implements the `Engine` methods) so that its presence was transparent to the component's clients (`EventBrokers` and `EventRouters`).

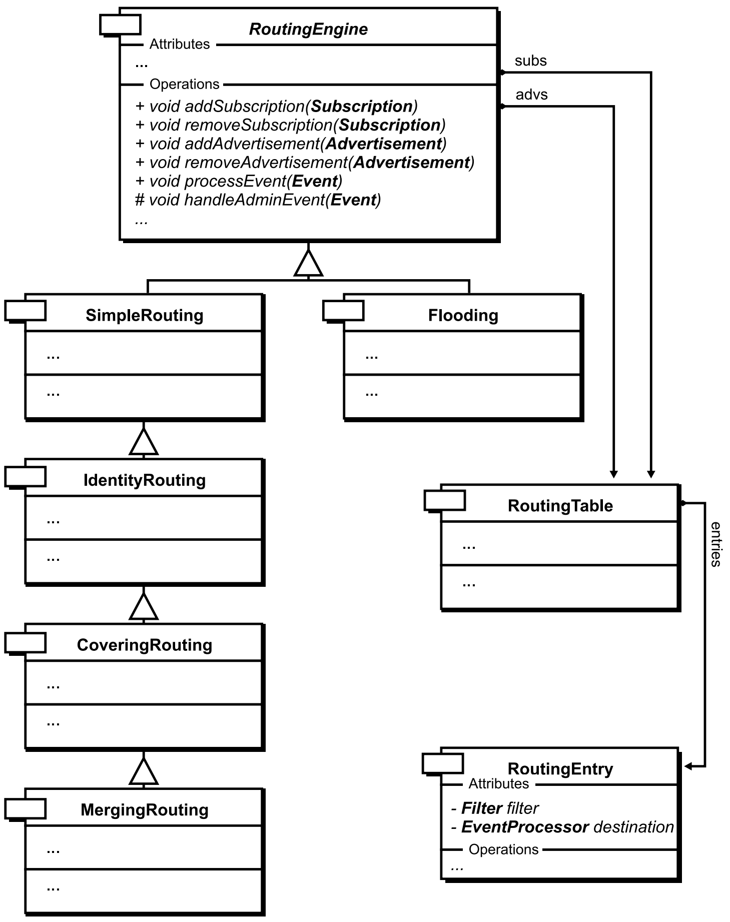

RoutingEngine
Attributes
...
Operations
+ void addSubscription(Subscription)
+ void removeSubscription(Subscription)
+ void addAdvertisement(Advertisement)
+ void removeAdvertisement(Advertisement)
+ void processEvent(Event)
# void handleAdminEvent(Event)
...
subs
advs
SimpleRouting
Flooding
IdentityRouting
CoveringRouting
MergingRouting
RoutingTable
entries
RoutingEntry
Attributes
- Filter filter
- EventProcessor destination
Operations

Figure 5.1: Rebeca's routing class hierarchy

The decorator forwards requests to the component and performs additional actions (before, after or both) in order to cache or deliver the incoming notifications or `Events`. Specific subclasses of `CachingEngine` were the ones who actually implemented the caching strategies, for instance in the `LocalBrokerCachingEngine` class. This scheme's transparency also allows for a dynamic addition and removal of an unlimited number of responsibilities at runtime.

While this approach promotes a separation of the concerns *routing efficiency* and *caching*, it presents at least a number of disadvantages, as well:

- The `Engine` hierarchy becomes quickly difficult to understand because now its purpose is twofold. The `CachingEngine` subclasses have names rather long which indicates a heavy composition of functionality.

- Client's code can't rely on object identity now, since from an object identity point of view, the decorator objects (the `CachingEngines`) are not identical as the decorated components themselves (the `RoutingEngines`).

- It is not clear how will the clients *transparently* handle `CachingEngine` objects, which is one of the requirements elicited in Section 4.1 (page 21).

  – Some of the source code of the clients classes must be edited in order to appropriately create instances of these classes. This produces code related to the caching concern that is not modularized (i.e., is spread around several classes and methods), is difficult to reason about and difficult to change.

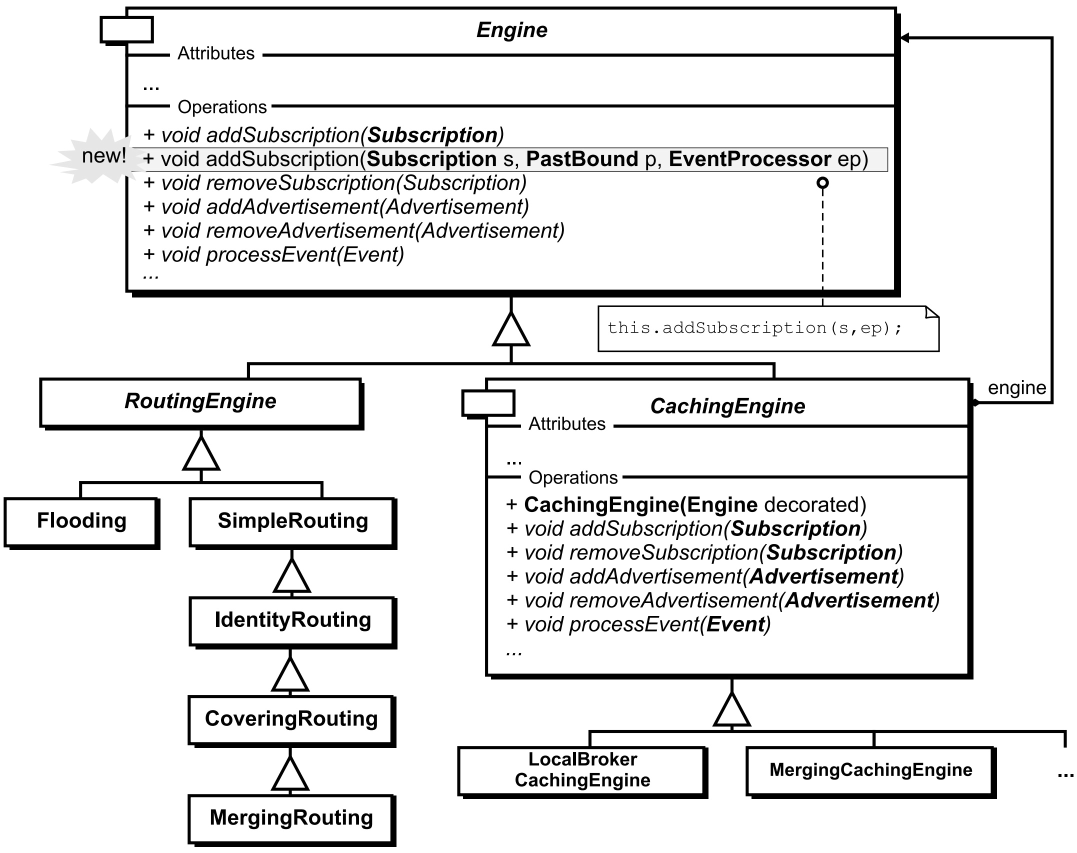

Engine
Attributes
...
Operations
+ void addSubscription(Subscription)
new!
+ void addSubscription(Subscription s, PastBound p, EventProcessor ep)
+ void removeSubscription(Subscription)
+ void addAdvertisement(Advertisement)
+ void removeAdvertisement(Advertisement)
+ void processEvent(Event)
...
this.addSubscription(s,ep);
RoutingEngine
Flooding
SimpleRouting
IdentityRouting
CoveringRouting
MergingRouting
CachingEngine
Attributes
...
Operations
+ CachingEngine(Engine decorated)
+ void addSubscription(Subscription)
+ void removeSubscription(Subscription)
+ void addAdvertisement(Advertisement)
+ void removeAdvertisement(Advertisement)
+ void processEvent(Event)
...
engine
LocalBroker CachingEngine
MergingCachingEngine
...

Figure 5.2: The *Decorator* design pattern approach

– The `RoutingEngine` class and its subclasses are packaged in the `rebeca.routing` package, which would be no longer an appropriate name for it since it will contain caching-related classes. Renaming this package to `rebeca.engine` involves editing the source code. Moreover, this package would contain only the `Engine` class, while two new subpackages `rebeca.engine.routing` and `rebeca.engine.caching` should be created as well, generating further changes.

- Finally, source code related to caching (e.g., the new `subscribe(..)` method) *crosscuts* source code related to routing. This led to an artificial `Engine` class that is a mixture of the needed methods of both concerns. Hence we are producing code that is not easy to understand nor maintain.

In the next section we will show how an architectural approach based on Aspect-Oriented Programming provides a solution to each of these disadvantages.

5.1.2 The AOP way

Aspect-Oriented Programming (AOP) mainly aim is to support *separation of concerns*. This means a way of modularizing crosscutting concerns much like Object-Oriented Programming is a way of modularizing common concerns. We refer the reader to Appendix A to get an introduction to aspect-oriented software development. In order to appropriately start working with AOP, we go back to the requirements stage. Here we start using the mechanisms proposed in [27] to handle non-functional requirements, by following three steps:

1. Crosscutting concerns: Identification of the non-functional requirements: the quality of *Response Time* crosscuts the *Routing Algorithm* concern. Hence, we've found what they call a

Crosscutting Concern	Caching
Description	Bootstrap latency of subscription process should be as minimal as possible (if not zero)
Priority	Max
List of Requirements	Performance, Response Time
List of Models	*Use cases:* SubscribesToInterest UnsubscribesFromInterest AdvertisesInformation UnadvertisesInformation PublishesNotifications GetsNotified

Table 5.1: Specification of the crosscutting concern *Caching*

candidate aspect or *aspect* for simplicity, and we will call it *Caching*. This aspect is specified using a template in Table 5.1.

2. Functional requirements: Traditional specification of functional requirements. This item has already been developed in Section 4.1 (page 21).

3. Composed requirements, which in turn consists of two parts:

   - First, composing functional requirements (modeled using UML) with aspects. With this purpose, three concepts define how can the composition be arranged:
     - *Overlapping*, when the requirements of the aspect modifies the functional requirements they transverse. In this case, the aspect requirements may be requested before the functional ones, or they may be requested after them.
     - *Overriding*, when the requirements of the aspect superpose the functional requirements they transverse. In this case, the behavior described by the aspect requirements substitutes the functional requirements behavior.
     - *Wrapping*, when the requirements of the aspect "encapsulate" the functional requirements they transverse. In this case, the behavior described by the functional requirements is wrapped by the behavior described by the aspect requirements.

     In our case, we've chosen to *wrap* the functional requirements, which is the more flexible (but also complex) option.
   - Second, try to resolve conflicts that may arise from the aspect composition process. Since there is only one aspect, *Caching*, no conflicts arise. Otherwise a decision would had to be made in terms of which crosscutting concern should have the maximum priority. Just as examples, other aspects within REBECA could be *Security*, *Transaction handling*, usage of *Advertisements*, etc.

This process yields the diagram of Figure 5.3, which is an evolved version of Figure 2.8 at page 15. As a result from this section, we proceed by describing how we arrived to the concrete design and implementation of the mentioned `Caching` concern.

5.2 Design and Implementation with AOP

A great challenge in the modeling, design and implementation phases is to concentrate at one problem at a time and abstract from others. Good design models only display the information

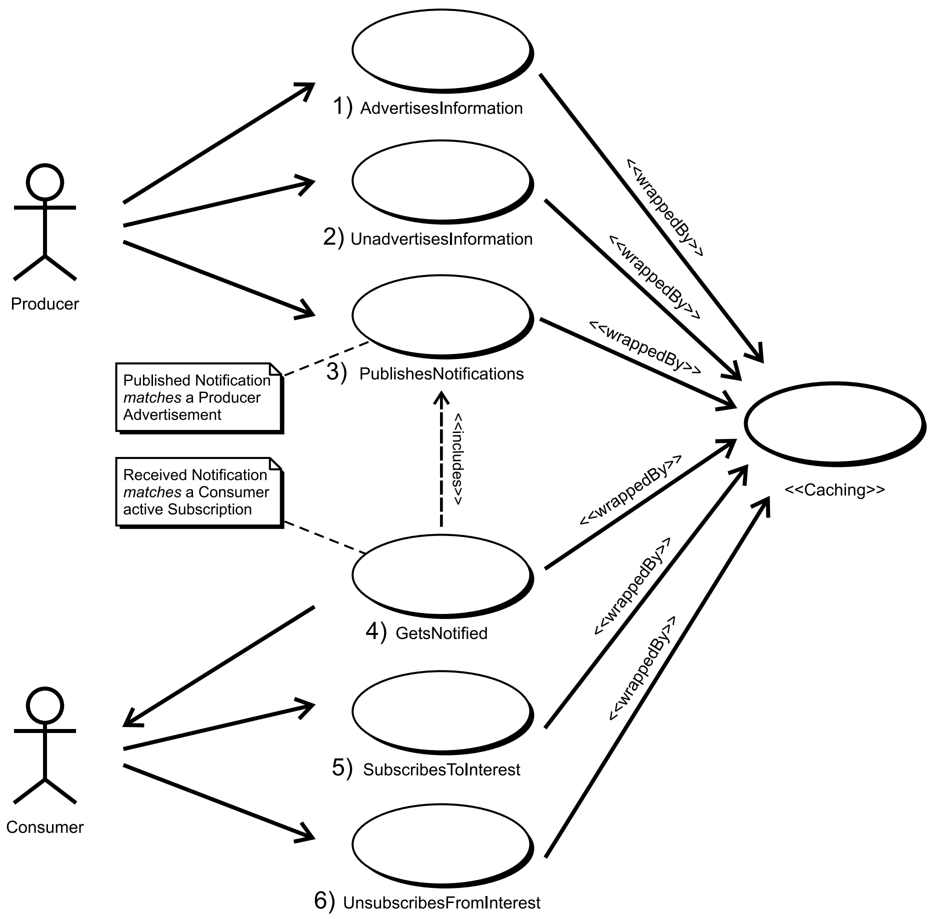

1) AdvertisesInformation
2) UnadvertisesInformation
3) PublishesNotifications
4) GetsNotified
5) SubscribesToInterest
6) UnsubscribesFromInterest
Producer
Consumer
<<Caching>>
<<wrappedBy>>
<<includes>>
Published Notification *matches* a Producer Advertisement
Received Notification *matches* a Consumer active Subscription

Figure 5.3: UML Use Case diagram involving the *Caching* aspect

essential for a specific purpose and abstracts from others. While AspectJ provides a suitable aspect-oriented *programming* language, no feasible standardized *modeling* language is at hand that supports the design of AspectJ programs, at the moment. Hence we have chosen different diagrams and UML extensions to further describe our ideas.

5.2.1 Caching Design

As suggested in the previous section, the focus of this stage is the design of the `Caching` aspect. Such a generic entity like the one shown in Figure 5.3, though, only provides superficial information on where and how (i.e., overlapping, overriding or wrapping) the aspect will participate. Actually, what we need is a model that provides a generic framework that allows different caching strategies to be implemented such as those described in the previous chapter.

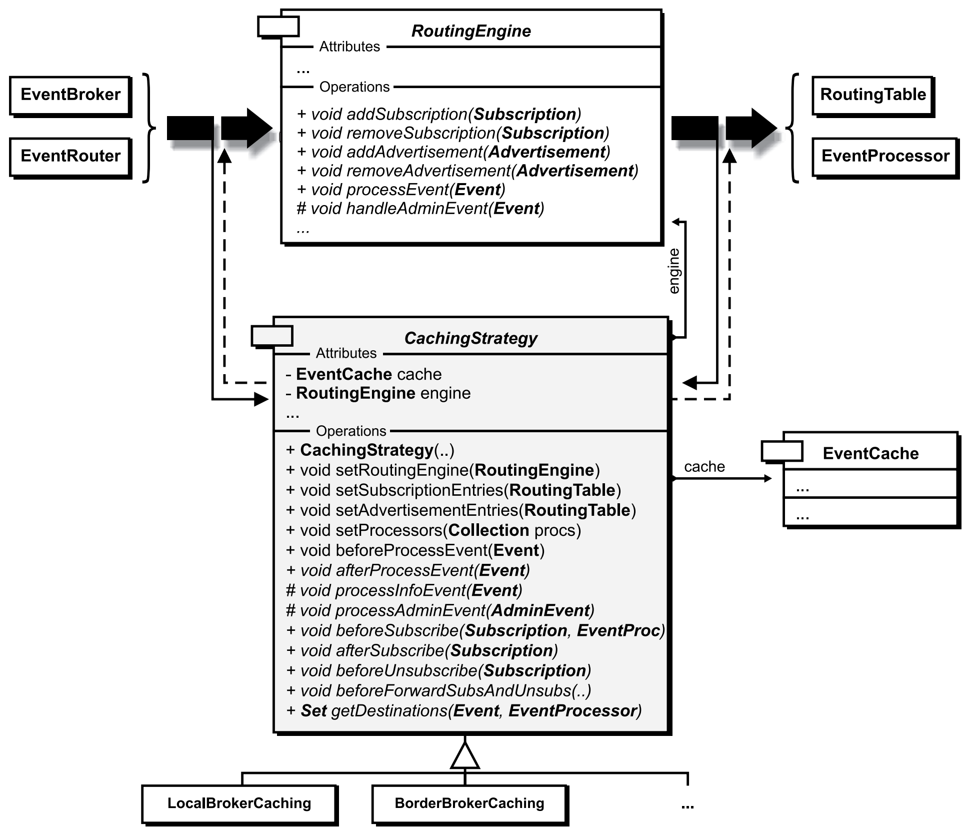
High Level Architecture

A common denominator that it's clear from the diagrams of Figure 4.4 to 4.20 is that the caching actions are related to the `RoutingEngine`'s activities, for instance:

- in reaction to subscribing and unsubscribing clients,
- in reaction to publishers advertising a filter or publishing an event,
- in reaction to brokers receiving administrative or data events, etc.

On one hand we must accept the non-avoidable high cohesion between the `RoutingEngine`s and the caching actions. In other words, caching actions *crosscut* the core routing functionality. On the other hand, in Section 5.1.1, we saw the drawbacks of pure object-oriented designs. Moreover, since the beginning a strong requirement was not to modify existent source-code, hence there was not even a chance to produce *tangling code.*

This led us to design a model that allows different caching strategies to be plugged. As a result, a combination of aspect-oriented facilities with object-oriented frameworks was used. Our approach is based in the non-invasiveness of AOP to allow a clear division of the base code from the new caching concern, while takes advantage of inheritance and polymorphism at the same time. The overall architecture is depicted in Figure 5.4. The diagram shows the `RoutingEngine` class, its clients (`EventBroker` and `EventRouter` classes) and its delegatees (`RoutingTable` and `EventProcessor` classes). Moreover, it shows that messages sent to and received from a `RoutingEngine` are somehow intercepted by an abstract specialization of it, called `CachingStrategy`. Here is where AOP comes into play. Note that in contrast with Figure 5.2, `CachingStrategy` subclasses' names are shorter and only represent what they are (in contrast to OOP where caching and routing would be mixed).



RoutingEngine
Attributes
...
Operations
+ void addSubscription(Subscription)
+ void removeSubscription(Subscription)
+ void addAdvertisement(Advertisement)
+ void removeAdvertisement(Advertisement)
+ void processEvent(Event)
# void handleAdminEvent(Event)
...
EventBroker
EventRouter
RoutingTable
EventProcessor
engine
CachingStrategy
Attributes
- EventCache cache
- RoutingEngine engine
...
Operations
+ CachingStrategy(..)
+ void setRoutingEngine(RoutingEngine)
+ void setSubscriptionEntries(RoutingTable)
+ void setAdvertisementEntries(RoutingTable)
+ void setProcessors(Collection procs)
+ void beforeProcessEvent(Event)
+ void afterProcessEvent(Event)
# void processInfoEvent(Event)
# void processAdminEvent(AdminEvent)
+ void beforeSubscribe(Subscription, EventProc)
+ void afterSubscribe(Subscription)
+ void beforeUnsubscribe(Subscription)
+ void beforeForwardSubsAndUnsubs(..)
+ Set getDestinations(Event, EventProcessor)
cache
EventCache
...
...
LocalBrokerCaching
BorderBrokerCaching
...

Figure 5.4: High level architecture

Next we go on with a detailed description of this idea by describing more precisely how AOP intervenes in the design.

Intercepting the Execution

The main effort of the modeling task is to identify a structure for the crosscutting concern by inspecting the routing behavior, in order to detect which are the *well-defined points in the program flow* [2] that need to be intercepted. A common task used by designers to perform this detection is to insert simple print lines of code at those points they are interested in (e.g., `System.out.println(..)` instructions). To achieve this in a more clear and elegant way, [2] proposes the usage of a *development aspect*. As it name indicates, development aspects are aspects that can be used during the development of an application. These aspects facilitate debugging, testing and performance tuning work. The aspects define behavior that ranges from simple tracing, to profiling, to testing of internal consistency within the application. They make it possible to cleanly modularize this kind of functionality, easily enabling and disabling the functionality when desired. Starting from a simple kind of aspect like that, we have progressively worked out the `Caching` aspect by selecting the crosscutting execution points (a.k.a. joinpoints) that a generic caching strategy needs to perform its actions. In order to identify these joinpoints for practical use, we have defined several *pointcuts*. A pointcut is like a named predicate on joinpoints that can match or not match any given joinpoint. Also, a pointcut can compose other joinpoints or pointcuts by means of the logical operators *and* (spelled `&&`), *or* (spelled `||`), and *not* (spelled `!`). This allows the creation of very powerful pointcuts from the simple building blocks of primitive pointcuts.

A major issue in designating these pointcuts is to define them the more precise as possible in order to prevent erroneous interceptions, e.g., by accidentally intercepting calls from unknown classes and methods. Hence we proceed by carefully describing and illustrating which these pointcuts are. We provide both a textual description as well as the expression in the AspectJ language that designates the pointcut. These pointcuts' list is as follows:

❶ The first pointcut that we need is related to the instantiation of `CachingStrategy` objects (Figure 5.5). Whenever a `RoutingEngine` object is *initialized* with a no-args constructor (i.e., `new()`), the execution will be intercepted and a `CachingStrategy` object is attached to it. The instantiation is performed by reflection on information provided by the environment. The pointcut is defined by the following expression:

```
pointcut initRoutingEngine():
            initialization(RoutingEngine.new());
```

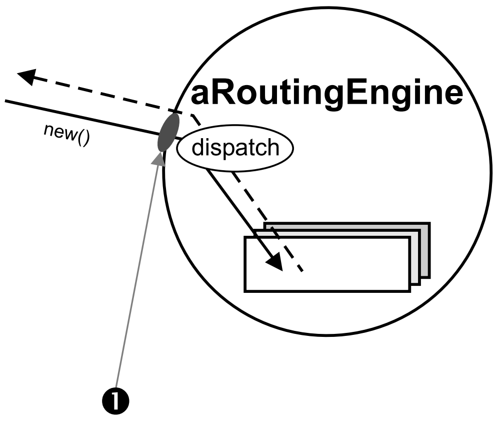


Figure 5.5: Pointcut to create `CachingStrategy` objects

❷, ❸ The next pointcuts' goal is to associate the subscriptions and advertisements `RoutingTable`s used by the `RoutingEngine` to the `CachingStrategy` (Figure 5.6). Caching strategies might need this information to take their decisions. Whenever the `RoutingEngine` instance variables fields named `subEntries` or `advEntries` are to be *set within* the *code* of the no-args constructor method, the execution is intercepted. A reference to the new value is passed to the `CachingStrategy` object, which preserves it. The pointcuts are defined by the following expressions:

```
pointcut initSubscriptionEntries():
            withincode(RoutingEngine.new()) && set(RoutingTable RoutingEngine.subEntries);

pointcut initAdvertisementEntries():
            withincode(RoutingEngine.new()) && set(RoutingTable RoutingEngine.advEntries);
```

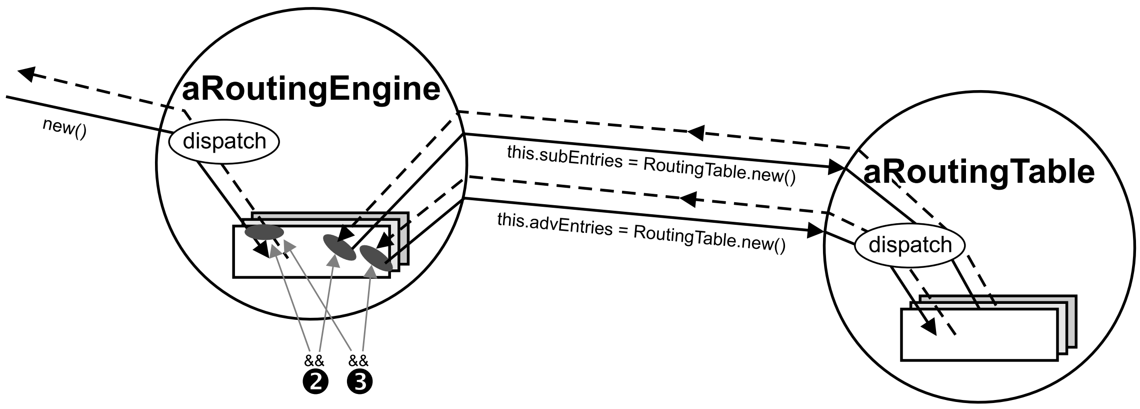

aRoutingEngine
new()
dispatch
this.subEntries = RoutingTable.new()
this.advEntries = RoutingTable.new()
aRoutingTable
dispatch
&&
&&
❷
❸

Figure 5.6: Pointcuts to pass the `RoutingTable`s to `CachingStrategy` objects

❹ Similarly to the previous pointcuts, the next pointcut's goal is to associate a broker's neighbors list to the caching object (Figure 5.7). This list is required when more complex routing algorithms are used by the brokers, in order to force forwarding caching requests to other neighbors not selected by the router. Whenever the `RoutingEngine` instance variable field named `procs` is to be *set within* the *code* of the no-args constructor method, the execution is intercepted. A reference to the new value is passed to the `CachingStrategy` object, which preserves it. The pointcut is defined by the following expression:

```
pointcut initProcessors():
            withincode(RoutingEngine.new()) && set(Collection RoutingEngine.procs);
```

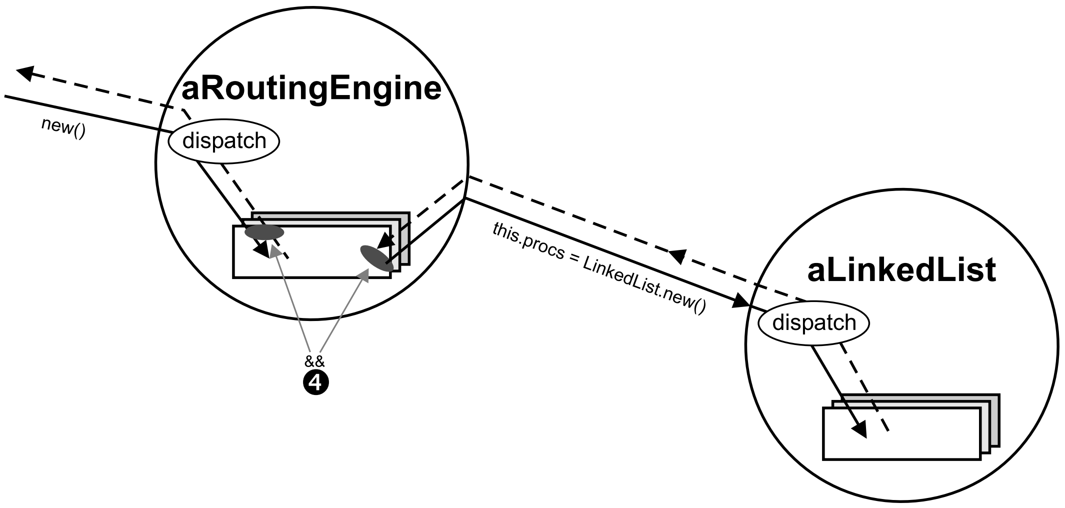


Figure 5.7: Pointcut to pass the neighbor `EventProcessor`'s collection to `CachingStrategy` objects

❺ The following pointcut crosscuts the subscription process (Figure 5.8). It is defined as the intersection between two joinpoints: the first of them is *within* the *code* of the `LocalEventBroker`'s `subscribe(..)` method; while the second is when the `RoutingEngine`'s `addSubscription(..)` method is *called* and returns or throws. Different caching strategies will perform different actions in this pointcut, so they are advised before and after it. The pointcut is defined by the following expression:

```
pointcut subscription():
            withincode(void LocalEventBroker.subscribe(Subscription,EventProcessor)) &&
            call(void RoutingEngine.addSubscription(Subscription, EventProcessor));
```

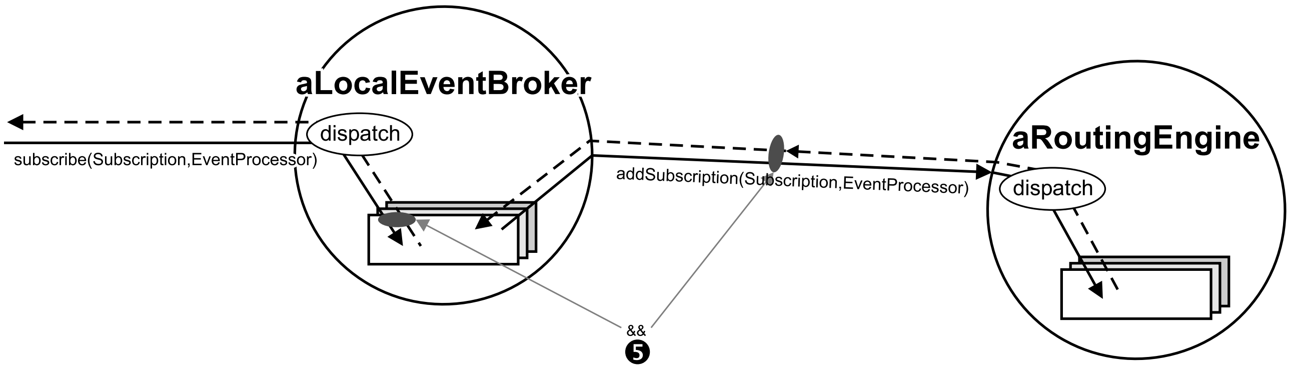

aLocalEventBroker
dispatch
subscribe(Subscription,EventProcessor)
addSubscription(Subscription,EventProcessor)
aRoutingEngine
dispatch
&&
❺

Figure 5.8: Pointcut to intercept *subscriptions*

❻ Analogically to the previous item, this pointcut crosscuts the unsubscription process (Figure 5.9). It is defined as the intersection between two joinpoints: the first of them is *within* the *code* of the `LocalEventBroker`'s `unsubscribe(..)` method; while the second is when the `RoutingEngine.removeSubscription(..)` method is *called* and returns or throws. The caching strategies might perform some activities in this pointcut, so they are advised here too. The pointcut is defined by the following expression:

```
pointcut unsubscription():
            withincode(void LocalEventBroker.unsubscribe(Subscription)) &&
            call(void RoutingEngine.removeSubscription(Subscription));
```

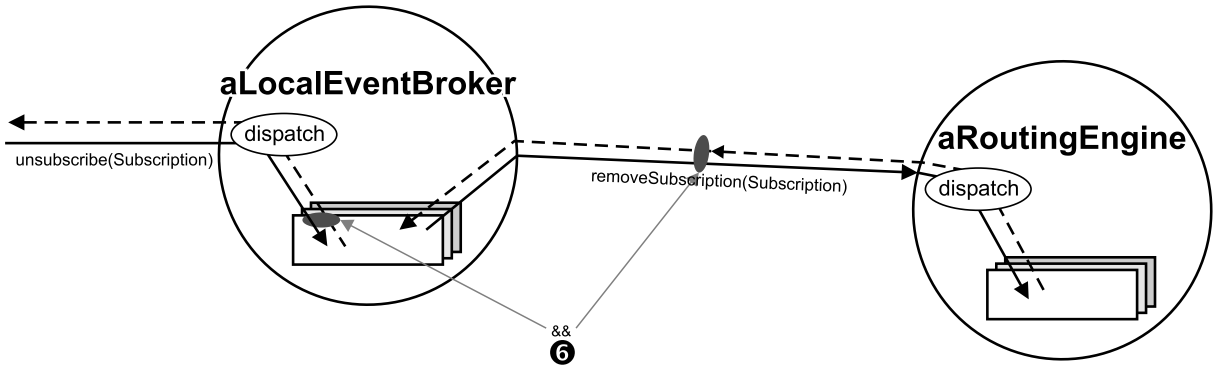


Figure 5.9: Pointcut to intercept *unsubscriptions*

❼ Clearly, the arrival of a message at a broker is an event of interest to the caching strategies. They might register it in their data structure for future consumers, perform administrative actions, or just discard it. The pointcut crosscuts the event processing by the routing algorithm (Figure 5.10). It is defined as the intersection between two joinpoints: the first of them is *within* a `RoutingEngine` anonymous `Thread`'s[1] `run()` method; while the second is when the `RoutingEngine`'s `processEvent2(..)` method is *called* and returns or throws. The pointcut is defined by the following expression:

```
pointcut eventProcessing():
            within(RoutingEngine) &&
            call(void RoutingEngine.processEvent2(Event, EventProcessor));
```

[1]This thread's duty is to repeatedly route events taken from a queue as they are asynchronously added into it. Despite being anonymous, we call it `RoutingEngineThread` in the diagram.

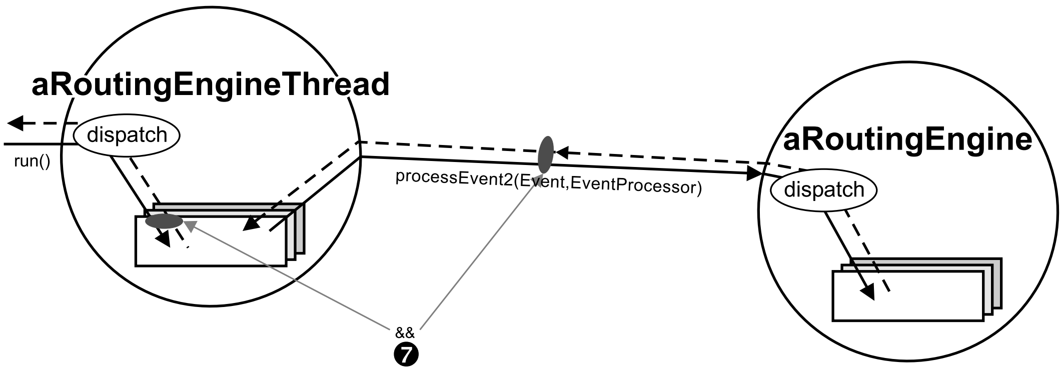


Figure 5.10: Pointcut to intercept *event processing*

❽ Another happening of interest for the caching strategies to react occurs when the routing algorithm forwards an administrative event containing subscriptions and unsubscriptions to a neighbor broker. The pointcut crosscuts this happening at the `SimpleRouting` class, since the `Flooding` algorithm doesn't forward administrative events (Figure 5.11). It is defined as the intersection between two joinpoints: the first of them is *within* the *code* of the `SimpleRouting`'s `forwardSubsAndUnsubs(..)` method; while the second is when the `EventProcessor`'s `process(..)` method is *called* and returns or throws. The pointcut is defined by the following expression:

```
pointcut forwardSubsAndUnsubs():
            withincode(void SimpleRouting.forwardSubsAndUnsubs(Vector, Vector, EventProcessor)) &&
            call(void EventProcessor.process(Event));
```

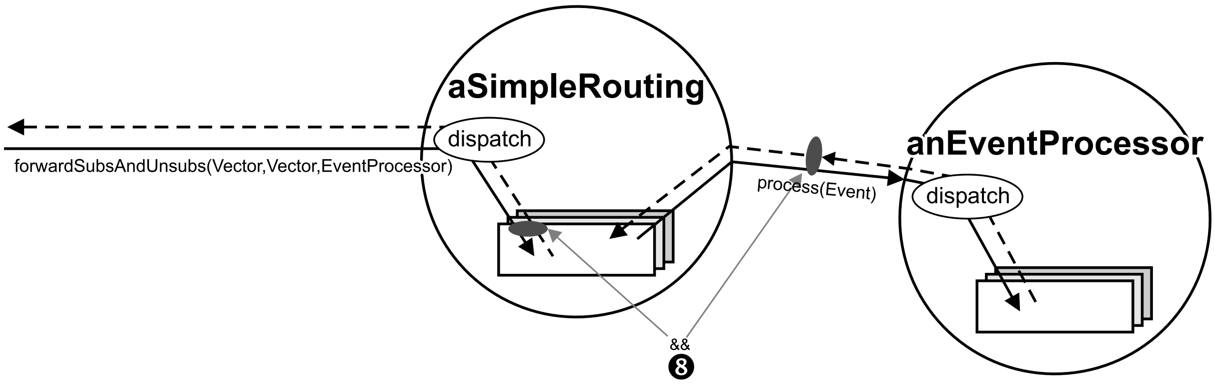


Figure 5.11: Pointcut for *subscriptions* and *unsubscriptions* forwarding

❾ Last but not least, the set of destinations chosen by the routing algorithm to forward an event to neighbors and/or clients is also important for a caching strategy. For instance, if the destination chosen for an event is currently blocked, it shouldn't receive the event but retain it. This is handled by the caching strategies accordingly. The pointcut crosscuts the destinations selection by the routing algorithm (Figure 5.12). It is defined as the intersection between two joinpoints: the first of them is *within* the *code* of the `RoutingEngine`'s `processEvent2(..)` method; while the second is when the `RoutingTable.getDestinations(..)` method is *called* and returns or throws. The pointcut is defined by the following expression:

```
pointcut getDestinations():
            withincode(void RoutingEngine.processEvent2(Event, EventProcessor)) &&
            call(Set RoutingTable.getDestinations(Event));
```

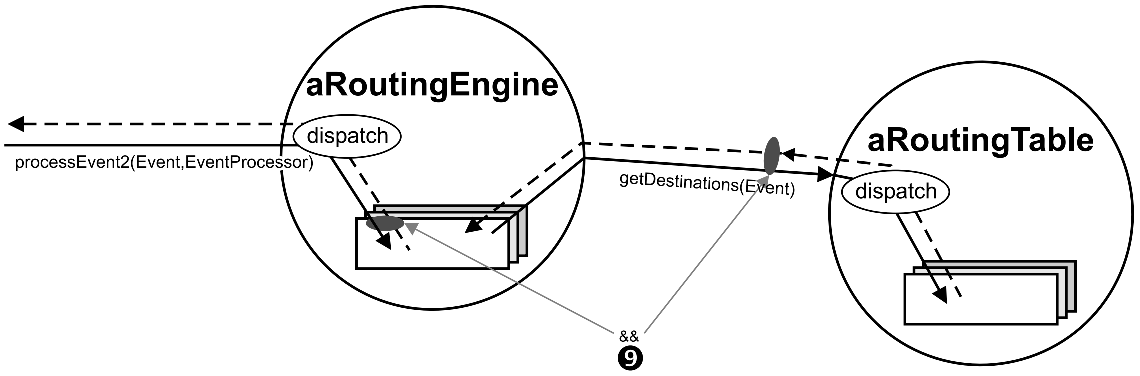


Figure 5.12: Pointcut for *event destination* selection

Note that these pointcuts pick out joinpoints, but they don't do anything apart from that. Actually, they demarcate where a caching strategy might be applied. To actually implement cross-cutting behavior, we use *advices*. An advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points). These pieces of code are the ones which implement the functionality of directing the execution towards the associated `CachingStrategy` object.

Introducing Type Members

We have deliberately avoided an important issue so far that is how to incorporate the new `subscribe(..)` method with the signature previously described in Figure 3.2 (page 19) to the `EventBroker`'s interface, without changing the source code. Again, AOP provides another powerful concept, the *open classes*. Within the AspectJ jargon, it is possible to use *inter-type declarations*. AspectJ's inter-type declarations (a.k.a. *introductions*) are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. Unlike the *advices*, which operate primarily dynamically, introductions operate statically, at compile-time.

We can declare the method necessary to implement the new caching capability, and associate it to the `EventBroker` interface by means of the following AspectJ sentence:

```
public void EventBroker.subscribe(Subscription sub,
                                  EventProcessor source,
                                  PastBound bound);
```

As expected for a standard interface method, each class that implements the `EventBroker` interface (i.e., the `LocalEventBroker` and `DefaultEventBroker` classes) *must* implement this method too, otherwise a compile-time error would be raised by the compiler. Thus, we also provide the implementations of the introduced method for each of these classes, accordingly. On one hand, a `DefaultEventBroker` doesn't truly implement a broker, but rather delegates all calls to a nested `EventBroker` instance, hence this behavior is similarly replicated. A `LocalEventBroker`, on the other hand, is the one who provides the real behavior. Here we simple attach the `PastBound` information into the `Subscription`, and again delegate it to the `subscribe(Subscription, EventProcessor)` normal method (which is intercepted later by the respective pointcut).

Provided that `Subscription` objects don't know anything about carrying `PastBound` objects, we define another set of introductions: a private field and associated setter/getter methods.

```
    private PastBound Subscription.bound = null;

    public void Subscription.setPastBound(PastBound bound) { this.bound = bound; }
```

```
public PastBound Subscription.getPastBound() { return this.bound; }

public boolean Subscription.isBounded() {  return this.bound != null; }

public void Subscription.removePastBound() { this.bound = null; }
```

With this set of declarations, it is possible to attach `PastBound` information to a `Subscription` object, without invading their source code.

In a similar fashion, `RoutingEntry` objects need to be able to block and retain, in a `Reply` object, those notifications received in the meantime while a broker requests a reply and it effectively arrives. Hence we define another set of crosscutting elements to implement the caching behavior related to the held routing entry. First, we define the necessary inter-type declarations which allow a `RoutingEntry` to have an associated `Reply`:

```
private Reply RoutingEntry.reply;

public void RoutingEntry.setReply(Reply reply) { this.reply = reply; }

public Reply RoutingEntry.getReply() { return this.reply; }

public void RoutingEntry.removeReply() { this.reply = null; }

public boolean RoutingEntry.isBlocked() { return this.reply != null && !this.reply.isBlocking(); }
```

Then, we define a pair of pointcuts that intercept the creation of `RoutingEntry` objects:

❶ When a client application issues a subscription, it is registered at its local broker. The pointcut (Figure 5.13) is defined as the intersection between two joinpoints: the first of them is *within* the *code* of the `RoutingEngine`'s `addSubscription(..)` method; while the second is when a `RoutingEntry`'s `new(Subscription, EventProcessor)` *initialization* method is called. The pointcut is defined by the following expression:

```
pointcut initSubscriptionRoutingEntry():
            withincode(void RoutingEngine.addSubscription(Subscription, EventProcessor)) &&
            call(RoutingEntry.new(Filter, EventProcessor));
```

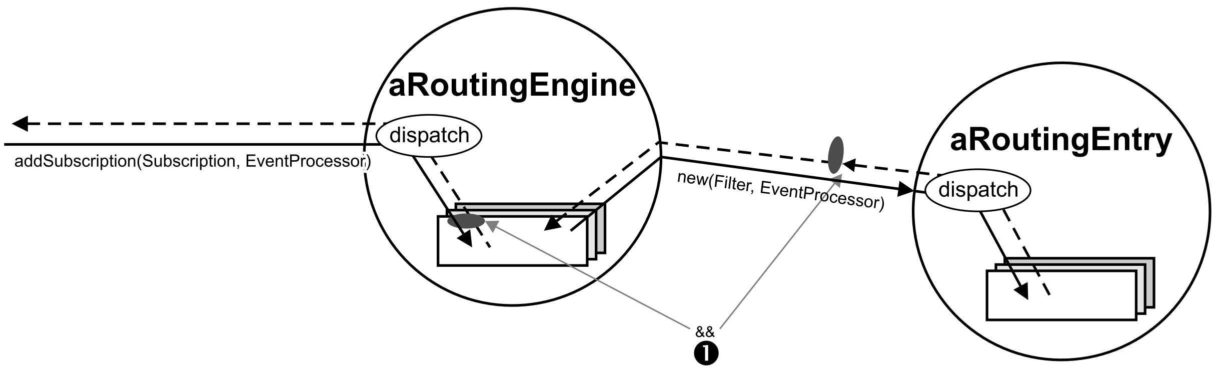

aRoutingEngine
dispatch
addSubscription(Subscription, EventProcessor)
new(Filter, EventProcessor)
aRoutingEntry
dispatch
&&
❶

Figure 5.13: Pointcut to *initialize* a `RoutingEntry` object

❷ When an administrative event arrives at an event router containing a collection of subscriptions and unsubscriptions, many routing entries are created simultaneously. The pointcut that crosscuts this behavior (Figure 5.14) is defined as the intersection between two joinpoints: the first of them is *within* the *code* of the `RoutingTable`'s `add(Collection, EventProcessor)` method; while the second is when the `RoutingEntry`'s `new(Filter, EventProcessor)` *initialization* method is called. The pointcut is defined by the following expression:

```
pointcut initForwardedSubRoutingEntry():
            withincode(void RoutingTable.add(Collection, EventProcessor))  &&
            call(RoutingEntry.new(Filter, EventProcessor));
```

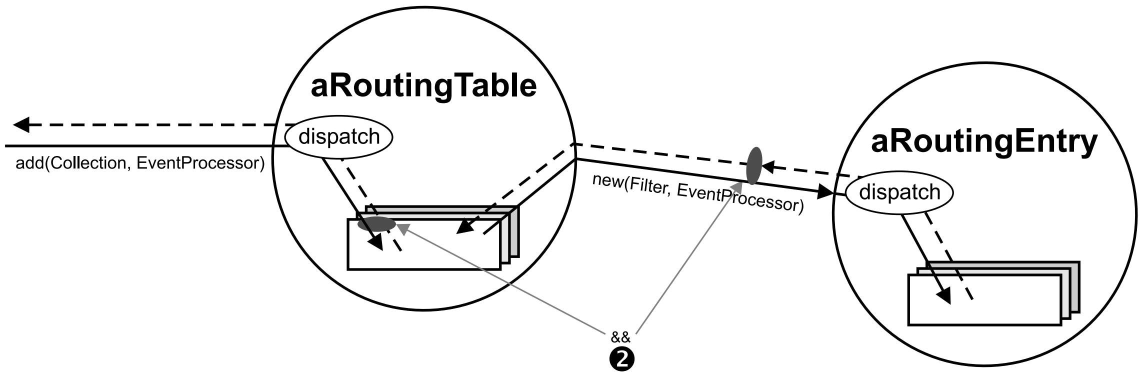


Figure 5.14: Pointcut to *initialize* many `RoutingEntry` objects

The behavior for these pointcuts is similar: when a `RoutingEntry` is about to be initialized with a `Subscription` that has `PastBound` information attached to it (which can be checked by calling the `isBounded()` method), a new `Reply` object is attached to the former. This is implemented by a couple of advice elements associated to each pointcut.

Finally, we also need a bidirectional reference between a `RoutingEngine` and its associated `CachingStrategy` object. Instead of directly declaring an inter-type field at the `RoutingEngine`, a slightly different design suggested in [2] was used. We have designed a `HasCachingStrategy` interface, which has a field of type `CachingStrategy`, and related setter/getter methods.

```
public interface HasCachingStrategy{
  private CachingStrategy cachingStrategy;

   public CachingStrategy getCachingStrategy()
   {
     return this.cachingStrategy;
   }

   public void setCachingStrategy(CachingStrategy cachingStrategy)
   {
     this.cachingStrategy = cachingStrategy;
   }
};
```

The main difference though relies in a special AspectJ declaration which states that any type that is a `RoutingEngine` implements the public interface `HasCachingStrategy`. In this way, it is possible to make any class implement the defined interface in a more flexible way.

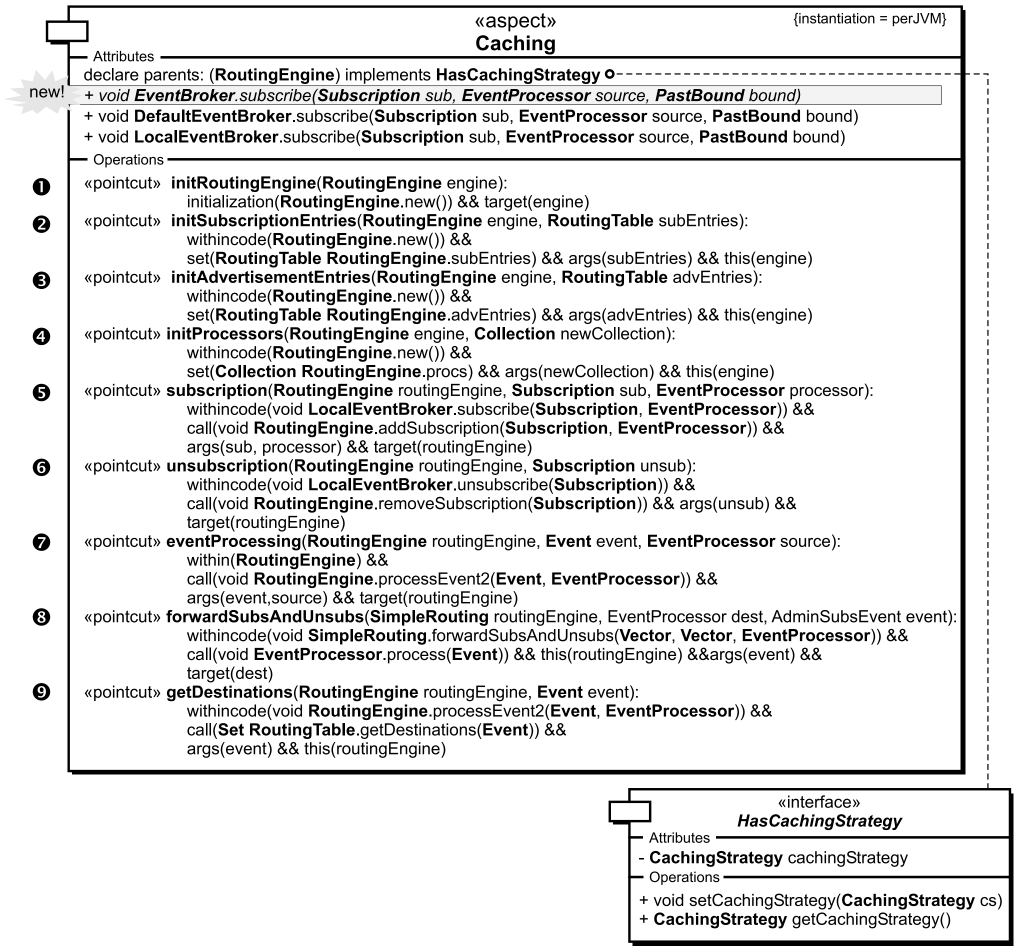
```
declare parents: (RoutingEngine) implements HasCachingStrategy;
```

Putting it all together

As it was mentioned before, AOP mainly aim is to support *separation of concerns*. However, this wouldn't be useful if it didn't provide means to modularize the concerns appropriately. In this regard, AOP introduces a new modular unit, called *aspect*, to serve as container for crosscutting elements. We have grouped the related pointcuts and declarations accordingly into respective aspects.

To illustrate these aspects, the Aspect-Oriented Design Model [34] proposes the representation of aspects as UML classes of a special stereotype (called ≪aspect≫, in imitation to AspectJ's aspects). The aspects are supplied with tagged values that specify how the aspects are to be instantiated. We have chosen to initialize aspects in a per-Java Virtual Machine basis.

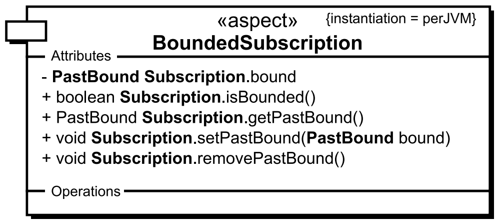
Pointcuts can also *expose* part of the execution context at their join points in order to be of practical use. Values exposed by a pointcut can be used in the body of advice declarations. Hence the pointcut definitions are accompanied by the needed exposed values. In Figures 5.15, 5.16 and 5.17, the `Caching`, `BoundedFilter` and `HeldRoutingEntry` aspects are shown, respectively.



```
«aspect»                                        {instantiation = perJVM}
Caching
Attributes
declare parents: (RoutingEngine) implements HasCachingStrategy
new! + void EventBroker.subscribe(Subscription sub, EventProcessor source, PastBound bound)
+ void DefaultEventBroker.subscribe(Subscription sub, EventProcessor source, PastBound bound)
+ void LocalEventBroker.subscribe(Subscription sub, EventProcessor source, PastBound bound)
Operations
❶ «pointcut»  initRoutingEngine(RoutingEngine engine):
                  initialization(RoutingEngine.new()) && target(engine)
❷ «pointcut»  initSubscriptionEntries(RoutingEngine engine, RoutingTable subEntries):
                  withincode(RoutingEngine.new()) &&
                  set(RoutingTable RoutingEngine.subEntries) && args(subEntries) && this(engine)
❸ «pointcut»  initAdvertisementEntries(RoutingEngine engine, RoutingTable advEntries):
                  withincode(RoutingEngine.new()) &&
                  set(RoutingTable RoutingEngine.advEntries) && args(advEntries) && this(engine)
❹ «pointcut» initProcessors(RoutingEngine engine, Collection newCollection):
                  withincode(RoutingEngine.new()) &&
                  set(Collection RoutingEngine.procs) && args(newCollection) && this(engine)
❺ «pointcut» subscription(RoutingEngine routingEngine, Subscription sub, EventProcessor processor):
                  withincode(void LocalEventBroker.subscribe(Subscription, EventProcessor)) &&
                  call(void RoutingEngine.addSubscription(Subscription, EventProcessor)) &&
                  args(sub, processor) && target(routingEngine)
❻ «pointcut» unsubscription(RoutingEngine routingEngine, Subscription unsub):
                  withincode(void LocalEventBroker.unsubscribe(Subscription)) &&
                  call(void RoutingEngine.removeSubscription(Subscription)) && args(unsub) &&
                  target(routingEngine)
❼ «pointcut» eventProcessing(RoutingEngine routingEngine, Event event, EventProcessor source):
                  within(RoutingEngine) &&
                  call(void RoutingEngine.processEvent2(Event, EventProcessor)) &&
                  args(event,source) && target(routingEngine)
❽ «pointcut» forwardSubsAndUnsubs(SimpleRouting routingEngine, EventProcessor dest, AdminSubsEvent event):
                  withincode(void SimpleRouting.forwardSubsAndUnsubs(Vector, Vector, EventProcessor)) &&
                  call(void EventProcessor.process(Event)) && this(routingEngine) &&args(event) &&
                  target(dest)
❾ «pointcut» getDestinations(RoutingEngine routingEngine, Event event):
                  withincode(void RoutingEngine.processEvent2(Event, EventProcessor)) &&
                  call(Set RoutingTable.getDestinations(Event)) &&
                  args(event) && this(routingEngine)

«interface»
HasCachingStrategy
Attributes
- CachingStrategy cachingStrategy
Operations
+ void setCachingStrategy(CachingStrategy cs)
+ CachingStrategy getCachingStrategy()
```

Figure 5.15: The `Caching` aspect

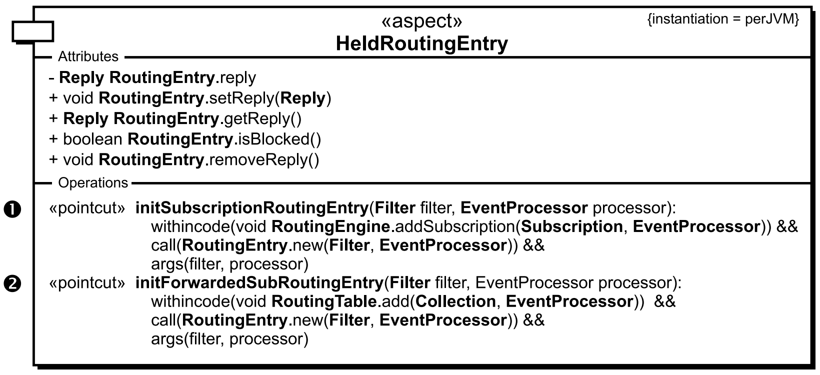


```
«aspect»              {instantiation = perJVM}
BoundedSubscription
Attributes
- PastBound Subscription.bound
+ boolean Subscription.isBounded()
+ PastBound Subscription.getPastBound()
+ void Subscription.setPastBound(PastBound bound)
+ void Subscription.removePastBound()
Operations
```

Figure 5.16: The `BoundedSubscription` aspect

5.2.2 Generic Behavior: Notification Storage

As it was depicted in Figure 5.4, the `CachingStrategy` class implements the notification storage by delegating this duty to another class called `EventCache`. This class merely implements the steps for



```
- Reply RoutingEntry.reply
+ void RoutingEntry.setReply(Reply)
+ Reply RoutingEntry.getReply()
+ boolean RoutingEntry.isBlocked()
+ void RoutingEntry.removeReply()

«pointcut»  initSubscriptionRoutingEntry(Filter filter, EventProcessor processor):
              withincode(void RoutingEngine.addSubscription(Subscription, EventProcessor)) &&
              call(RoutingEntry.new(Filter, EventProcessor)) &&
              args(filter, processor)
«pointcut»  initForwardedSubRoutingEntry(Filter filter, EventProcessor processor):
              withincode(void RoutingTable.add(Collection, EventProcessor))  &&
              call(RoutingEntry.new(Filter, EventProcessor)) &&
              args(filter, processor)
```

Figure 5.17: The `HeldRoutingEntry` aspect

the data structure defined in Section 4.3 (page 23). The notification storage design (Figure 5.18) is very simple and consists of a unidirectional relationship between the `EventCache`, `EventQueue` and `CountedItems` classes. Their responsibilities are the following:

- The `EventCache` class is responsible for managing the overall data structure. Its objects are initialized by specifying three parameters: `MaxGlobalBuffers`, `MaxIndexedFilters` and `MaxFilterBuffers`. These integer values define the size of the event cache. With this information, an index is created where each entry is composed of a `Subscription` and a `LinkedList`, where received matching events are enqueued. This class also addresses the allocation, enqueuing and dequeuing of notifications into the associated lists, and ensures that the size is preserved.

- The `EventQueue` class is a simple wrapper for a `LinkedList` that adds operations for behaving like a bounded size queue (restricted by the `MaxGlobalBuffers` value specified at the instantiation).

- The `CountedItem` class is a generic object wrapper that provides methods that aids in the process of having a reference counter for the wrappee.

5.2.3 Specialization of the Framework

In this section we illustrate how the caching strategies presented in the previous chapter are mapped to the design. We have already illustrated the generic *hooks* that the `Caching` aspect provides. Further, a generic `CachingStrategy` class was introduced in Figure 5.4 that implements the behavior that is common to all caching strategies. Each strategy is encapsulated in its own class, forming a class hierarchy depicted in Figure 5.19. Through the refinement mechanism of inheritance, each class specializes the behavior that is necessary to implement each strategy. Next we briefly describe the responsibilities of each of the participating classes.

CachingStrategy

`CachingStrategy` is the base class of all caching strategies: it defines a skeleton of abstract methods that each subclass must implement in reaction to the happenings that would be interesting for them. Not all its methods are abstract, though. When a `CachingStrategy` receives an `Event`, it is inspected to see if it's an administrative event or a data event, and then lets subclasses handle each type accordingly.

An object of the `CachingStrategy` class is initialized with three parameters that define the size of the associated `EventCache`. After initializing it, the `RoutingEngine` object that will be

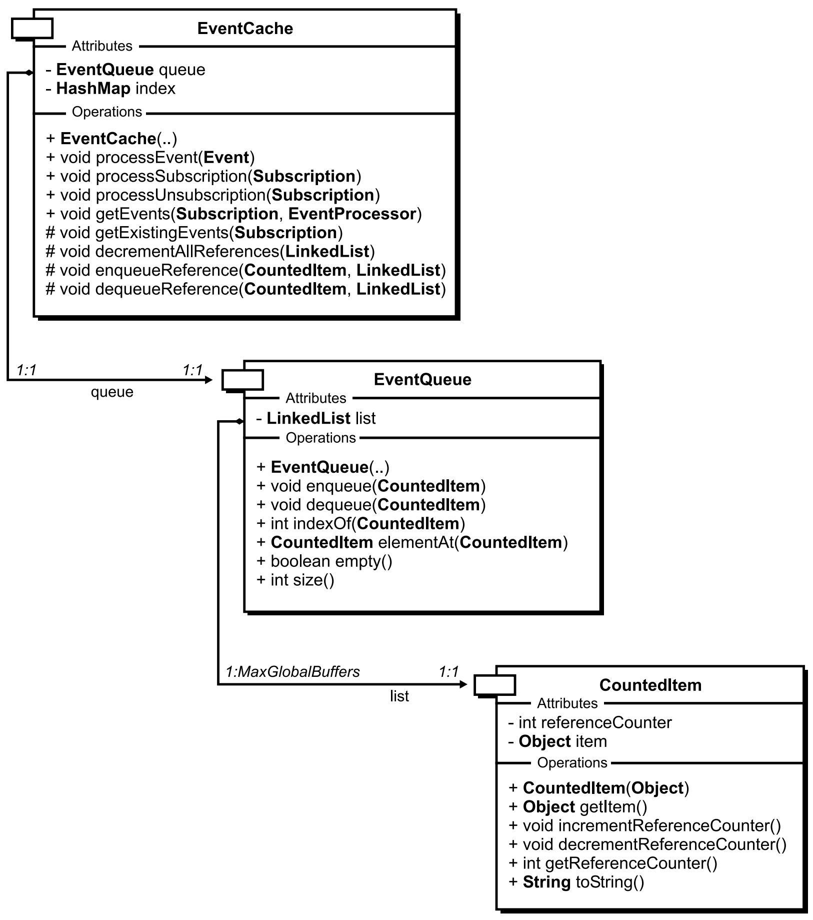

EventCache
Attributes
- EventQueue queue
- HashMap index
Operations
+ EventCache(..)
+ void processEvent(Event)
+ void processSubscription(Subscription)
+ void processUnsubscription(Subscription)
+ void getEvents(Subscription, EventProcessor)
# void getExistingEvents(Subscription)
# void decrementAllReferences(LinkedList)
# void enqueueReference(CountedItem, LinkedList)
# void dequeueReference(CountedItem, LinkedList)
1:1
1:1
queue
EventQueue
Attributes
- LinkedList list
Operations
+ EventQueue(..)
+ void enqueue(CountedItem)
+ void dequeue(CountedItem)
+ int indexOf(CountedItem)
+ CountedItem elementAt(CountedItem)
+ boolean empty()
+ int size()
1:MaxGlobalBuffers
1:1
list
CountedItem
Attributes
- int referenceCounter
- Object item
Operations
+ CountedItem(Object)
+ Object getItem()
+ void incrementReferenceCounter()
+ void decrementReferenceCounter()
+ int getReferenceCounter()
+ String toString()

Figure 5.18: The `EventCache` and `EventQueue` classes

monitored by the `CachingStrategy` is passed to the former, as well as its subscriptions and advertisements `RoutingTables` and the neighbor `EventProcessor`'s `Collection`. Given that the `RoutingEngine` it monitors processes incoming notifications serially and in FIFO-order, same does the `CachingStrategy`.

The `CachingStrategy` class is extended by the following classes that implement the corresponding homonymous caching strategies: `LocalBrokerCaching`, `BorderBrokerCaching` and `MergingCaching` (see Fig. 5.19).

LocalBrokerCaching

A `LocalBrokerCaching` object encapsulates the caching strategy described in Section 4.4.1 (page 28). Within this strategy, data `Events` are only stored at the `LocalEventBroker`. As such, the strategy only works in conjunction with a deployment of the `Flooding` routing algorithm, which doesn't forward the subscriptions and unsubscriptions into the broker's network. Hence, this strategy neither needs to handle administrative events.

The main work resides *before* a subscription is about to be processed by a `RoutingEngine`. The strategy extracts the `PastBound` information attached to the issued `Subscription`, which serves as a constrain in the search of buffered notifications. The received `Subscription` is processed by the caching data structure (i.e., indexed if it doesn't contain an identical one, etc.). Whether enough or not to fulfill the request, locally stored events that match the filter are delivered back to the client. Then, the subscription process proceeds as normal, i.e., simply registering an appropriate `RoutingEntry` at the local `RoutingTable`.

Additionally, *before* an unsubscription issued by a client is about to be processed by the `RoutingEngine`, it is processed by the caching data structure, disposing the respective entry.

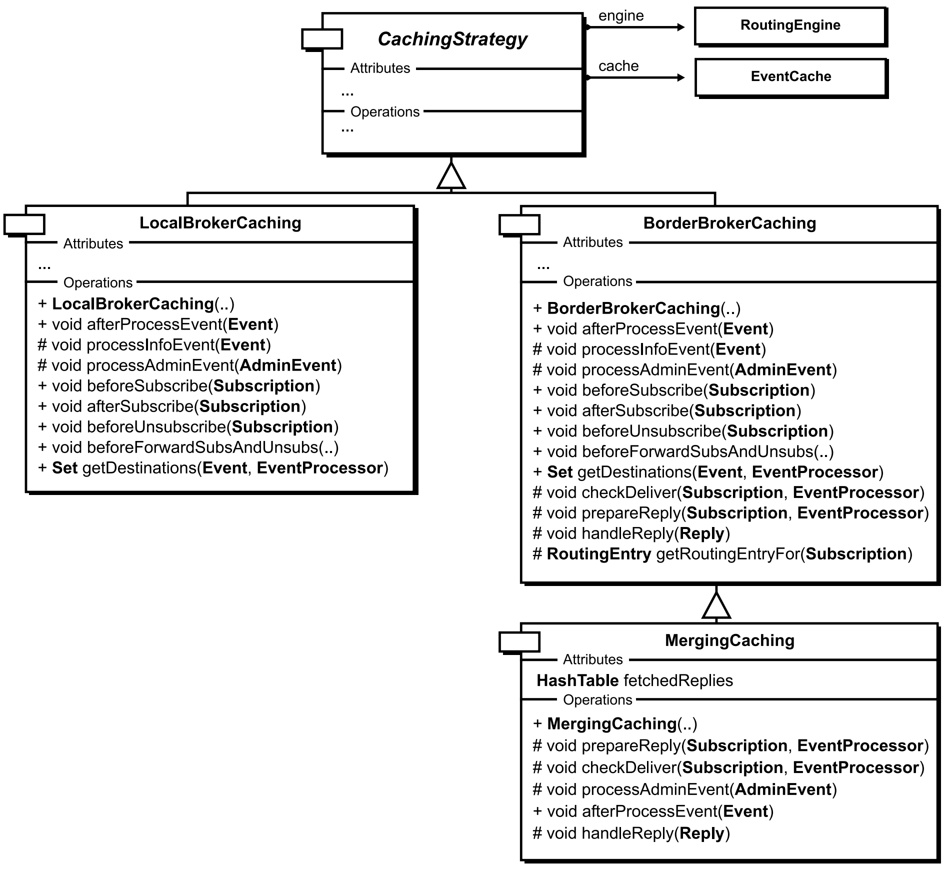

CachingStrategy
Attributes
...
Operations
...
engine
RoutingEngine
cache
EventCache
LocalBrokerCaching
Attributes
...
Operations
+ LocalBrokerCaching(..)
+ void afterProcessEvent(Event)
# void processInfoEvent(Event)
# void processAdminEvent(AdminEvent)
+ void beforeSubscribe(Subscription)
+ void afterSubscribe(Subscription)
+ void beforeUnsubscribe(Subscription)
+ void beforeForwardSubsAndUnsubs(..)
+ Set getDestinations(Event, EventProcessor)
BorderBrokerCaching
Attributes
...
Operations
+ BorderBrokerCaching(..)
+ void afterProcessEvent(Event)
# void processInfoEvent(Event)
# void processAdminEvent(AdminEvent)
+ void beforeSubscribe(Subscription)
+ void afterSubscribe(Subscription)
+ void beforeUnsubscribe(Subscription)
+ void beforeForwardSubsAndUnsubs(..)
+ Set getDestinations(Event, EventProcessor)
# void checkDeliver(Subscription, EventProcessor)
# void prepareReply(Subscription, EventProcessor)
# void handleReply(Reply)
# RoutingEntry getRoutingEntryFor(Subscription)
MergingCaching
Attributes
HashTable fetchedReplies
Operations
+ MergingCaching(..)
# void prepareReply(Subscription, EventProcessor)
# void checkDeliver(Subscription, EventProcessor)
# void processAdminEvent(AdminEvent)
+ void afterProcessEvent(Event)
# void handleReply(Reply)

Figure 5.19: The `CachingStrategy` class hierarchy

BorderBrokerCaching

A `BorderBrokerCaching` object encapsulates the caching strategy described in Section 4.4.2 (page 31). Within this strategy, data `Events` are only stored at the Border Broker. This strategy only works in conjunction with a deployment of the `SimpleRouting` routing algorithm, which ensures that every (un)subscription will be forwarded to the Border Broker[2].

Since data events are stored at the Border Broker's `EventRouter`, also replies are assembled there. Replies are objects of class `Reply`, which in turn is a subclass of the `AdminEvent` class. A `Reply` object might contain an arbitrary number of data `Events` requested by a client. In the meantime between a reply is requested and it effectively arrives, a blocking mechanism is performed at the `LocalEventBroker`. This blocking mechanism guarantees that the bootstrapping delay of a consumer is no larger than without the caching functionality in the worst and better in the average case.

Several helper methods aid in the process of supporting this behavior, such as the merging of the received reply with the notifications received in the meantime, the preparation of the reply itself, etc.

MergingCaching

A `MergingCaching` object encapsulates the caching strategy described in Section 4.4.3 (page 35). In its pure version, this strategy works in conjunction with a deployment of the `SimpleRouting` routing algorithm.

In contrast with the previous strategies that extracted the `PastBound` information from the `Subscription` at the Local Event Broker or Border Broker (respectively), this strategy permits the

[2]In fact, `SimpleRouting` floods (un)subscriptions across the network's brokers.

attached information to flow along with the subscription forwarding process in order to allow other Inner Brokers to register the request and cache matching notifications. This further complicates the reply mechanism since an `EventRouter` might ask several neighbor brokers for a reply, hence this must be properly managed. With this purpose, each queried broker is registered in the held reply. Later as their replies arrive, they are merged in a first-come-first-serve basis. When the last reply arrives (or alternatively, if there are no neighbor brokers to query), the overall reply is delivered back where it comes from. In the case that the request comes from a neighbor's link, it is delivered back as a `Reply` containing the matching notifications, otherwise (i.e., the request came from a consumer's link) the contained notifications are delivered one by one.

Moreover, this class also provides the caching behavior described in Section 4.4.4 (page 38). Hence, other (more complex) routing algorithms can be used in conjunction with this strategy. When the strategy is initialized, a boolean `ForceForwards` parameter is passed. As its name indicates, it specifies whether or not to force the forwarding of requests to neighbor brokers not chosen by the routing algorithm (e.g., `CoveringRouting`). In such case, to each non-selected neighbor a `FetchEvent` message is forwarded that explicitly indicates the request for matching notifications. The `FetchEvent` class is also a subclass of the `AdminEvent` class, and it simply carries the `Subscription` object for which events are requested, as well as the `PastBound` information. When an `EventRouter` receives an event like this, it simply registers the subscription in the caching data structure and prepares the reply accordingly.

Since this class shares much of the behavior of the `BorderBrokerCaching` class, the former extends the latter instead of extending the `CachingStrategy` class directly and having to rewrite (or scavenge) shared code.

5.3 Caching Strategies: Implementation Details

In contrast to traditional, from scratch, object-oriented designs, applying AOP is a little more complex. Furthermore, when source code modification is not desired, the implementation task gets even more difficult. Besides thinking in terms of types, variables, objects, inheritance and polymorphism, we must think in terms of *the existent source* (e.g., adequate encapsulation through getter and setter methods) and how to combine it using the joinpoint model that the aspect-oriented programming language offers. This is why the richness of the joinpoint model is a key issue in selecting an aspect-oriented *implementation* language in order to have the greatest possible flexibility. In this section we describe some of the decisions that drove the implementation phase.

5.3.1 Packaging with Aspects

Originally, within Rebeca, package names are unusual: instead of e.g. `org.tud.dvs.rebeca` only `rebeca` is used. Moreover, inner packages have been nested that group cohesive classes. For instance, `rebeca.routing` groups the routing algorithms, while `rebeca.network` groups low level event transport classes.

In order to accommodate the separation of aspects and components, in [5] it is suggested the usage of UML packages. Each aspect is encapsulated within its own package, and all the functionality of the aspect can be modeled within the package. The big picture is shown in Figure 5.20. For simplicity, only the most important packages are shown. The package `rebeca` groups the whole system, and inner packages group semantically close elements that tend to change together. We have imitated this grouping methodology and created the `rebeca.caching` package (which contains the previously described classes), as well as the inner `rebeca.caching.eventcache` package (which contains the caching data structure-related classes).

The package diagram also indicates that the aspects will crosscut components in the main system at certain join points. Circles with a cross inside (to indicate their cross-cutting nature) indicate the joinpoints. Thus the most relevant definitions of the join points are contained in brackets close to joinpoints.

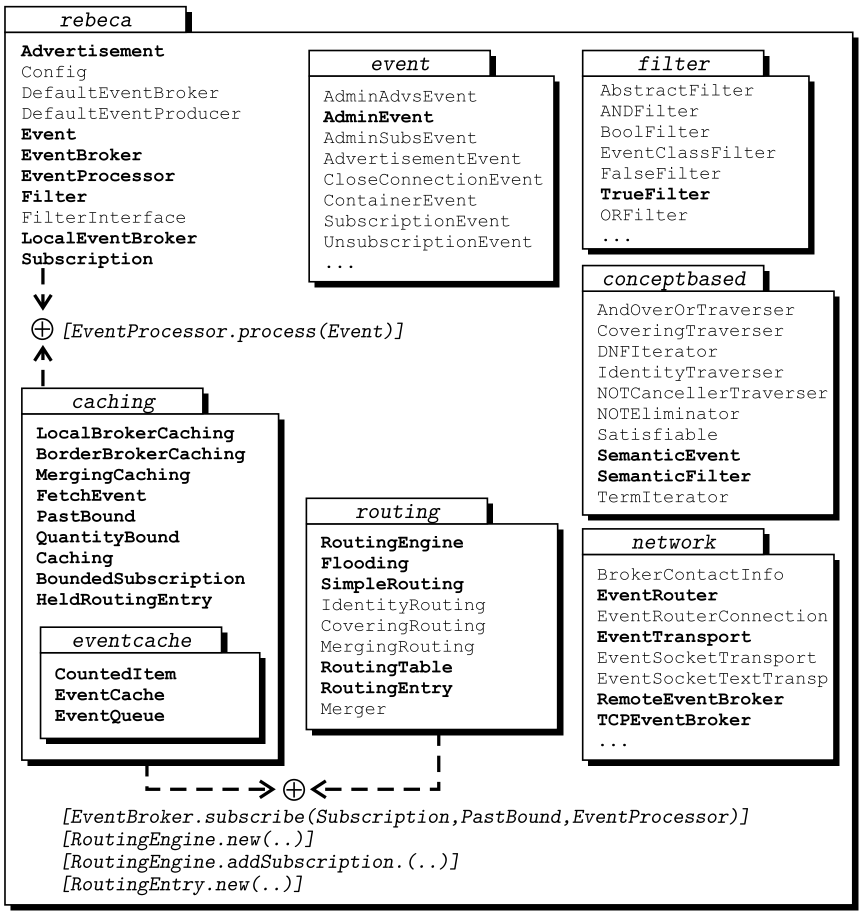


Figure 5.20: UML package diagram extended for aspects

5.3.2 Plug-and-Play Caching

One of the issues in starting the system using an aspect, other aspect, a set of aspects, or none, is *how* to specify at startup-time which classes to load, since aspect's source code is mapped to *bytecode* class files. Please note that within the AspectJ community this is another topic not yet explored in depth.

First we must clarify that the system is compiled with an AspectJ's own compiler, `iajc`, instead of the traditional `javac` Java compiler. At compile time, the compiler reads a `".lst"` text file which enumerates the source code files to be compiled. Hence, in principle, the primary way to specify which classes to build is to have several ".lst" files and rebuilding the system with the appropriate list. Our implementation contains two additional files:

- `plain.lst`, which lists whole system's files without the `rebeca.caching` package (and its `rebeca.caching.eventcache` package)
- `aspects.lst`, which lists whole system's files with the `rebeca.caching` package (and its `rebeca.caching.eventcache` package)

With this compilation mechanism (despite being somewhat precarious), AspectJ made it possible to cleanly modularize the caching functionality, and easily enable or disable the functionality when desired. In practice, to further automatize the usage of these files, an ANT script was written that allows to select which `".lst"` file to use from the command line.

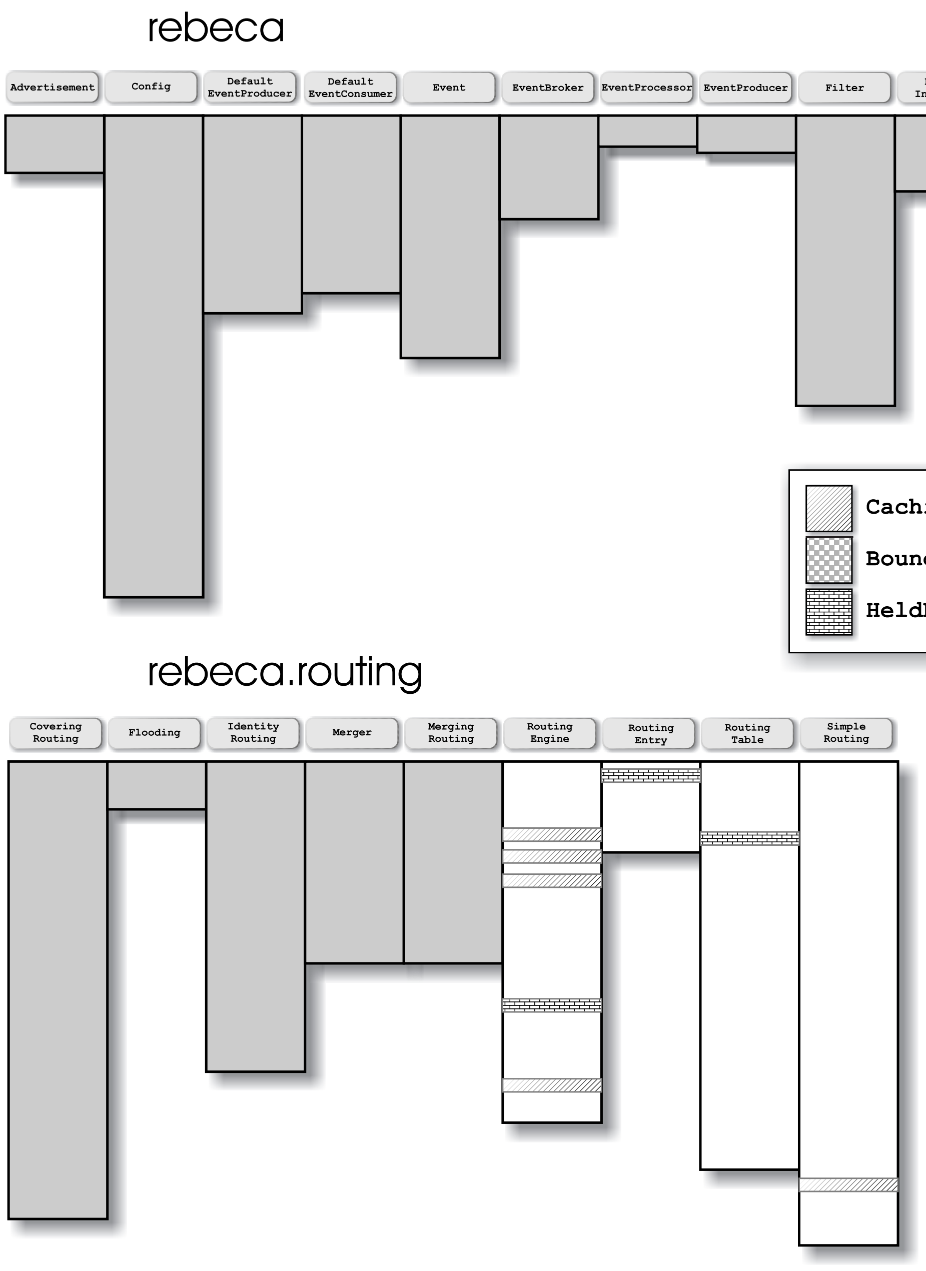


Figure 5.21: Aspect visualizer weaving points

5.4 Summary

As mentioned previously, one of the major hindrances was to devise the `Caching` aspect with the restriction of not modifying the existent source code. This is because there is no way (i.e., privilege) for an aspect to access the value of a class' private variable or invoke a private method. Hence, if the source was not adequately encapsulated, alternative (and therefore, not so clean) accessing mechanisms must be used. Luckily, Rebeca, and particularly the `RoutingEngine` class was pretty well designed and implemented. Therefore we only had to deal with this issue at a few points.

From the aspect-oriented design and implementation point of view, we can conclude the following benefits:

- In spite of being a crosscutting concern, we have achieved a good modularity. Since the beginning we knew that *caching* was not a completely orthogonal functionality. Nevertheless it contributed in the separation of concerns, e.g., by letting us think about the notification service *with or without* the caching functionality.

- The preexistent objects are not responsible for *caching*, because the `Caching` aspect encapsulates that responsibility. Their classes contain no calls to `CachingStrategy` methods, the `Caching` aspect encapsulates those calls.

- Routing objects have no knowledge of *caching*, hence they are shielded from changes to the `CachingStrategy` interface. Only the `Caching` aspect and `CachingStrategy` classes are affected. If the `CachingStrategy` base class changes, there is no need to modify the preexistent object classes, only the `Caching` aspect needs to be modified.

- Removing the *caching* functionality from the design is trivial: simply removing the `Caching` aspect and the `CachingStrategy` classes (or, what is the same, the `rebeca.caching` package).

- Removing the *Caching* functionality from the notification service implementation is trivial. The *Plug-and-Play* feature allows the `Caching` aspect to be unplugged without deleting it. Compiling with or without the aspect turns caching on or off, without editing classes and with no runtime cost. This also saves debugging code between uses.

Finally, it is worth mentioning a word about the implementation platform. Several were analyzed that support aspect-oriented software development (e.g., Borland's JBuilder plugins and AspectJ AJBrowser). Nevertheless, the one that is getting the most attraction in the community is the AspectJ Development Tools (AJDT) project. AJDT is a set of plugins for the open source platform Eclipse that provide support for aspect-oriented software development using AspectJ within the Eclipse IDE. Some of its strengths are keyword highlighting, AspectJ code templates, a powerful visualizer that shows the crosscutting impact of the aspects (we have replicated this information in Figure 5.21), support for multiple build configurations within a project, and, most important, a graphical structural members and crosscutting relationships display. This set of tools aid developers in the design and implementation stages of a system comprising aspects.

Chapter 6

Experimental Results

This chapter presents an evaluation of the implemented caching strategies presented in the previous chapter. The evaluation focuses on the characteristics of the caching strategies instead of system-specific parameters like CPU load or network bandwidth, etc. The goal here is to provide more details on their behavior.

6.1 General Setup

This section describes the general setup of the experiments which were performed in the context of a simulation of the enterprise application introduced in Section 1.2 (page 2). Here, clients (e.g., salesmen) subscribe and unsubscribe to certain currency exchange information, while several publishers of this information (e.g., financial organizations) exist. Besides the deployment configuration used (i.e., routing algorithm, usage of advertisements, caching strategy, etc.), the results are influenced by the characteristics of the broker topology, the consumers, and the producers. Investigating the relations among all these parameters is beyond the scope of this work. However, this evaluation varies some main parameters (e.g., the number of active subscriptions) and assumes a simple but meaningful scenario in which the other parameters remain constant (see Table 6.1). In the following subsections, the setup with respect to the mentioned parameters is described in more detail.

6.1.1 Broker Topology

The broker topology used has a major impact on any experiment. The main parameters that characterize a broker topology are:

Number of event routers	13
Number of local event brokers	22
Number of consumers per local broker	3 (∴ 66 total)
Number of subscriptions per consumer	△ 1 - 500
Number of possible currency exchanges	△ 1 - 10
Number of past notifications required	1
Number of notification sources	△ 1 - 10
Number of neighbor brokers	fix
Number of hierarchy levels	4
Assignment of subscriptions to consumers	random
Maximum Global Buffers	151
Maximum Indexed Filters	15
Maximum Filter's Buffers	10

Table 6.1: Fixed an varied parameters of the setup

- the number of brokers
- the number of neighbor brokers which may be constant or vary,
- the existence or absence of connectivity cycles, and
- the diameter of the network which is the longest path connecting two arbitrary brokers.

In line with Rebeca's evaluations, this work concentrates on a hierarchical, symmetrical, and acyclic, i.e., tree-like, topology. To use a symmetrical topology facilitates the interpretation of the findings, and the hierarchical structure is justified by the hierarchical structure of real networks, like the Internet.

The tested topology has 4 levels of brokers. Starting from a single router, called the *root router*, all routers except the leaves are connected to exactly 3 subordinate routers. To each non-leaf router, a single local broker is connected, while to each leaf router, two local brokers are connected, therefore, the used topology consists of 13 routers and 22 local brokers. In Figure 6.1 the topology is shown. The circles refer to routers while the squares refer to local brokers.

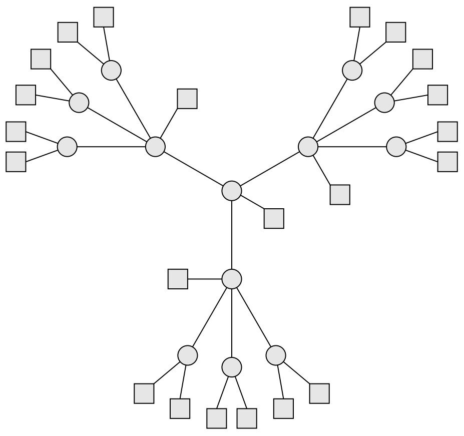


Figure 6.1: Broker topology with 4 levels

6.1.2 Characteristics of the consumers

The characteristics of the simulated consumers have a large impact on the evaluation of the caching strategies. Their main parameters are:

- the total number of consumers,
- the total number of subscriptions,
- the assignment of the subscriptions to the consumers (e.g., locality of interests),
- the number of notifications from the past requested with a subscription,
- the assignment of the consumers to the local brokers, and
- the rate of subscribing and unsubscribing.

In the experiments, the consumers are equally distributed among the local brokers in a fixed relation of 3:1. The evaluation randomly chooses a consumer and a subscription type (which is described later in Section 6.1.4), and the subscription is issued. This behavior is further repeated 500 times in order to have a good approximation of the selected parameters. Every subscription is issued together with a `QuantityBound` of only 1 notification from the past. Moreover, each consumer's subscription remains active until all of them are issued. At the end, they are all unsubscribed.

6.1.3 Characteristics of the producers

The characteristics of the producers also influence the caching behavior. The main parameters of a set of producers are:

- the absolute number of producers,
- the assignment of the advertisements to the producers,
- the assignment of the producers to the local brokers, and
- the publishing rate.

In the experiments, the number of launched producers varied according to the number of subscription types in a fixed relation of 1:1. The evaluation randomly chooses a local broker for a producer to attach to, and it starts publishing events that match the specified subscription type. They also initially issue an advertisement if the routing algorithm enforces so. The published exchange rate values are also randomly chosen in the interval 3±2, nevertheless they are clearly not of relevance.

6.1.4 Characteristics of the subscriptions and events

The characteristics of the subscriptions and events (i.e., the data model) used by subscribers and publishers also impact the results of the evaluation. Their main parameters are:

- the type of subscriptions, and
- the distribution of the subscriptions.

Our evaluations took advantage of the concept-based addressing model. Hence, ontologies were used to represent both subscriptions and events. In the currency exchange scenario we deal with subscriptions which express the interest for a given currency conversion, for instance, from *US dollars* to *Pesos Argentinos*. These currencies can be represented with a `Currency` ontology object, which is a unit of measure for the monetary dimension. In turn, a currency can be expressed in a number of ways, for example with its ISO 4217 three-letter code or a more complete, full currency name. This meta-information is specified in an associated `SemanticContext`. For simplicity, only conversions *from* US dollars to other currencies are considered. The evaluation distributes the subscriptions to consumers in a random and uniform way.

Each subscription is encapsulated in a `SemanticFilter` object. On the other hand, producers publish information in the form of `SemanticEvents`, which encapsulate (in this case) an `ExchangeRate` ontology object. The latter is further composed of a `FromCurrency`, a `ToCurrency`, a `Value`, and a `DateTime` (all of them self-explained).

6.2 Caching Efficacy

Although several metrics can be used in order measure the caching efficacy, we will concentrate on one that allows to observe with more detail how the bootstrap latency is reduced when caching is applied. The measure is simple: when a client issues a subscription with a given `PastBound`, the system monitors whether the request from the past was fulfilled or not. With this information we can effectively measure the percentage of subscriptions the notification service is able to attend entirely (which we will call "caching efficacy" in this chapter). For instance, when a client issues a subscription with a given `QuantityBound` of 1 notification and the system effectively delivers the required notification from the past, it logs a special message (`"1.0"`). Otherwise, it logs another special message (`"0.0"`). This stream of special messages is then extracted from the log to form the graphics of the current section.

With the extracted log information, other tables are constructed that reflect the percentage of subscriptions entirely attended so far. The tables are used to construct associated histograms. These histogram's horizontal axes represent the number of issued subscriptions (also as we move right along the h-axis we shift forward in time), whereas the vertical axis represents the caching efficacy. The histograms are displayed in the Figures from 6.2 to 6.6, and we have structured them from two points of view. First, they are arranged by caching strategy (Figures 6.2, 6.3, and 6.4), with different series that represent the extracted log data for several numbers of subscription types (it can be seen 1, 2, 3, 4, 5 and 10 subscription types). Then, they are arranged by number of subscription types (Figures 6.5 and 6.6), with several series that represent the caching strategies. This arrangement of the graphics allows us to compare the strategies and their behavior at the same time.

It is interesting to observe how the caching efficacy gets better as the system "warms-up". Provided that subscriptions are not revoked until the end of the simulation, it is easy to infer that the caching efficacy (measured as explained before) can only provide increasing values. This is of course only achieved in the long run (i.e., for a large number of issued subscriptions), after removing the initial random noise. The difference lies on how fast does the caching strategy start offering a certain overall quality of service. Other things can also be inferred from these graphics which we describe in the following subsections.

6.2.1 Local Broker Caching

As it can be seen in Figure 6.2, the caching efficacy (i.e., the number of subscriptions that the notification service entirely attends) depends on the number of possible subscription types. The more subscription types are considered, the less the notification service is able to attend the issued subscriptions. For instance, with the first 100 subscriptions and only one subscription type issued by clients, the caching efficacy grows to 55%. This performance cannot be maintained, however, when more subscription types are considered: with 4 subscription types only a 4% caching efficacy is achieved, whereas with 5 subscription types the system hasn't helped bootstrapping to even 1 client.

As we said previously, in the long run all the strategies will somehow contribute in the bootstrapping sequence, though the question is how fast should we expect the overall system to arrive a given efficacy (i.e., with how many subscriptions).

It must be noted also that the whole series were completed, i.e., the 500 subscriptions could be issued. This is because the `LocalBrokerCaching` strategy runs in conjunction with the `Flooding` routing algorithm, which is the simplest one.

6.2.2 Border Broker Caching

The next experiment involved the `BorderBrokerCaching` strategy. Its evaluation results are shown in Figure 6.3. In the same way as the previous strategy, the caching efficacy depends on the number of possible subscription types. However, and as it was expected, moving the data storage point inside the broker's network provides a better efficacy than storing the events at the local brokers. This is because more client's interests are embraced by the imaginary cones at the border brokers than at the local brokers (cf. Figure 4.11, page 32). For example, when 100 subscriptions were issued with 1 subscription type, the efficacy has grown to 72%, whereas with 5 subscription types the efficacy has grown to 12% (in contrast to the 0% of the previous strategy).

The chart of Figure 6.3 also strengths the idea of the long-run increasing caching efficacy value. Moreover, because of the randomness of the simulation, the chart shows the maximum initial noise in the 4 subscription types series. The reader should also note that the simulation could not be completed for all the series. In fact, as more subscription types are considered, less subscriptions

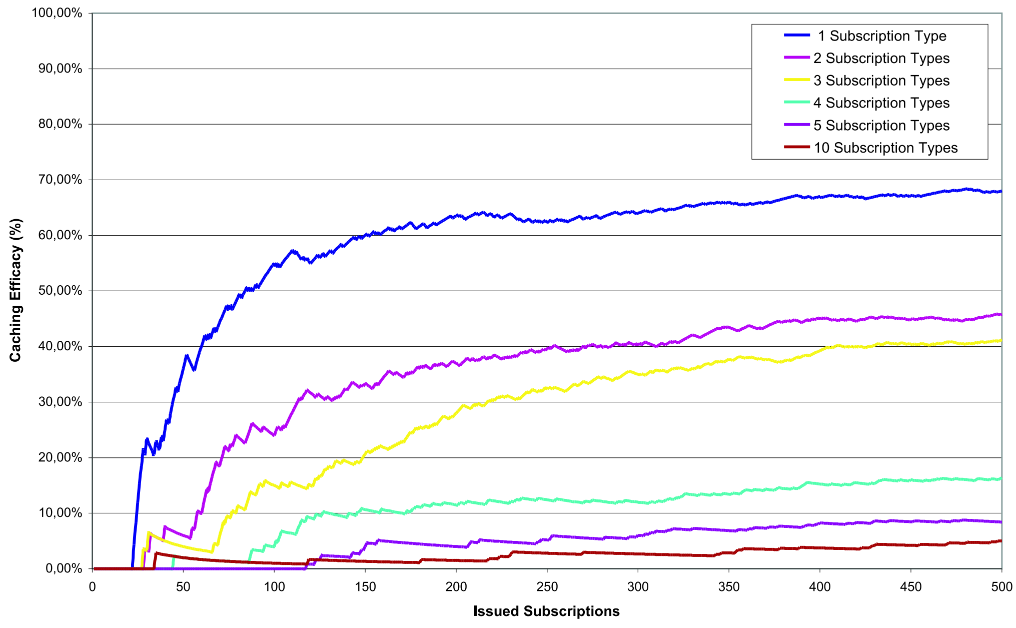


Figure 6.2: Local Broker Caching Evaluation

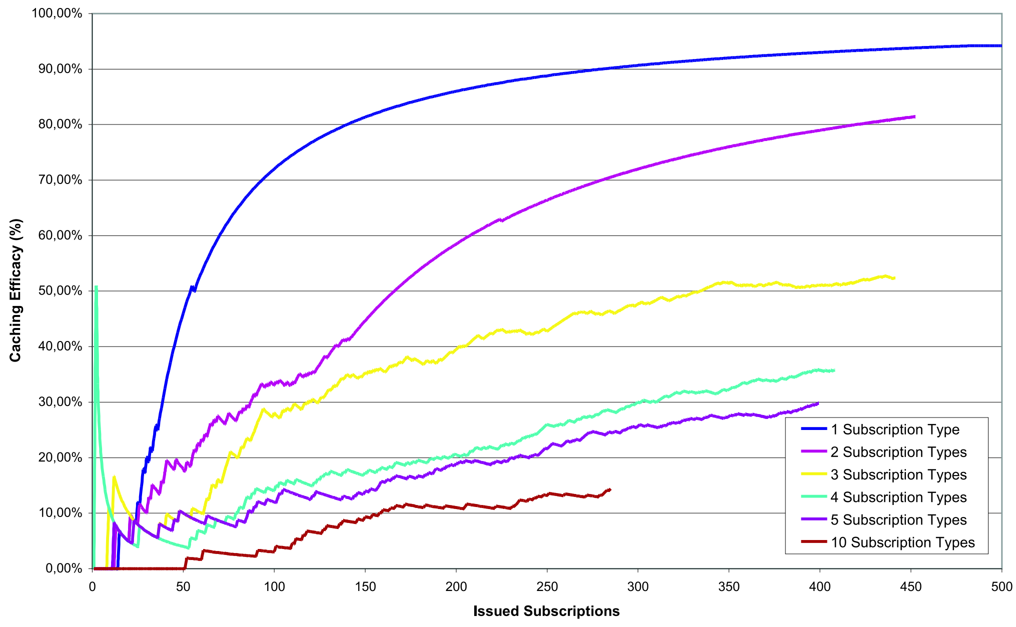


Figure 6.3: Border Broker Caching Evaluation

could be issued. This is in part because of the processor and physical memory requirements of the simulation, and in part because of the `SimpleRouting` filter forwarding algorithm used to perform the tests (which involves flooding the subscription across the broker's network). This heavier routing/caching combination behavior implies more network traffic and more helper objects than with the previous one, which is reflected also in the time spent to finalize the tests. We will get back to this subject later in the chapter's summary.

6.2.3 Merging Caching

The last measures taken were related to the `MergingCaching` strategy, which is depicted in Figure 6.4. At first sight the graphic doesn't contribute too much, since the 6 series are overlapping each other. Nevertheless, the fact that these series overlap has a very important consequence: the caching efficacy is independent of the number of subscription types. This behavior was expected since, as it was previously mentioned (cf. Section 4.4.3, page 35), the strategy's goal is to make it independent of the neighbor's interests. In contrast with the other strategies whose caching efficacy depends on the probability that a similar subscription has been issued earlier *at the local or border broker*, in this strategy the efficacy only requires that a similar subscription has been issued earlier *anywhere*. Hence this behavior rapidly augments the caching efficacy to almost a 100%, independently of the neighbor client's interests and (more important) the number of possible subscription types.

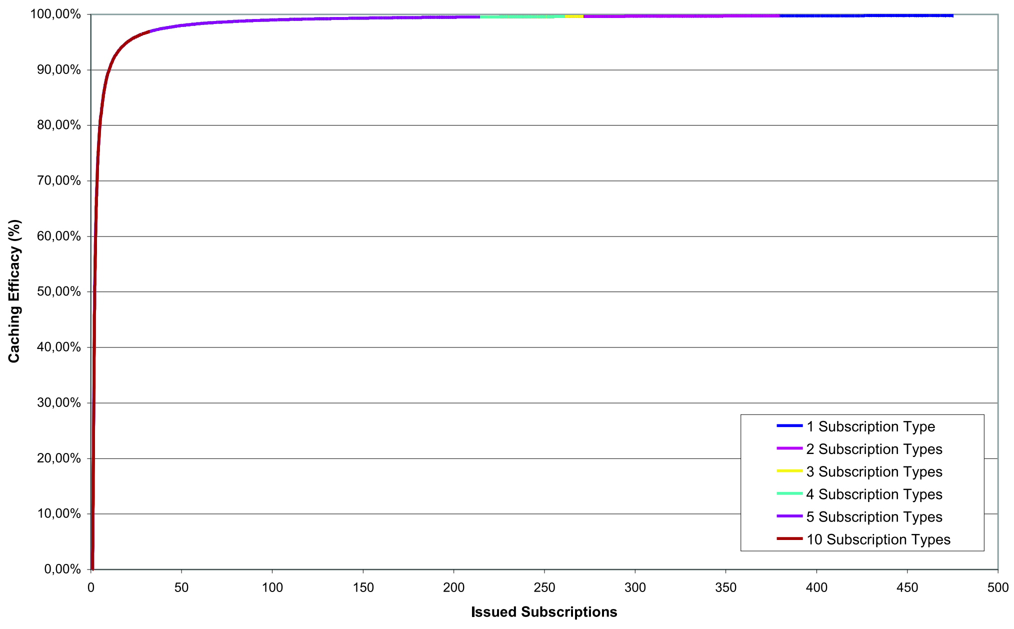


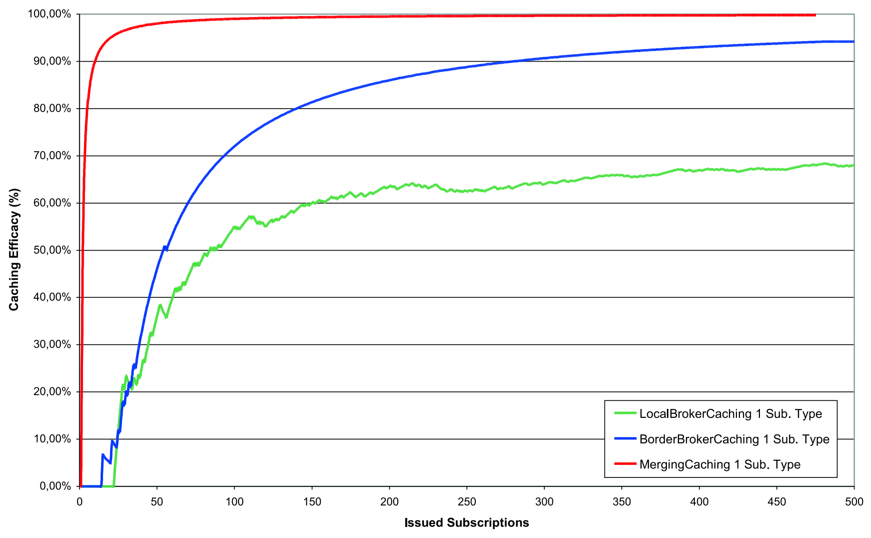
Figure 6.4: Merging Caching Evaluation

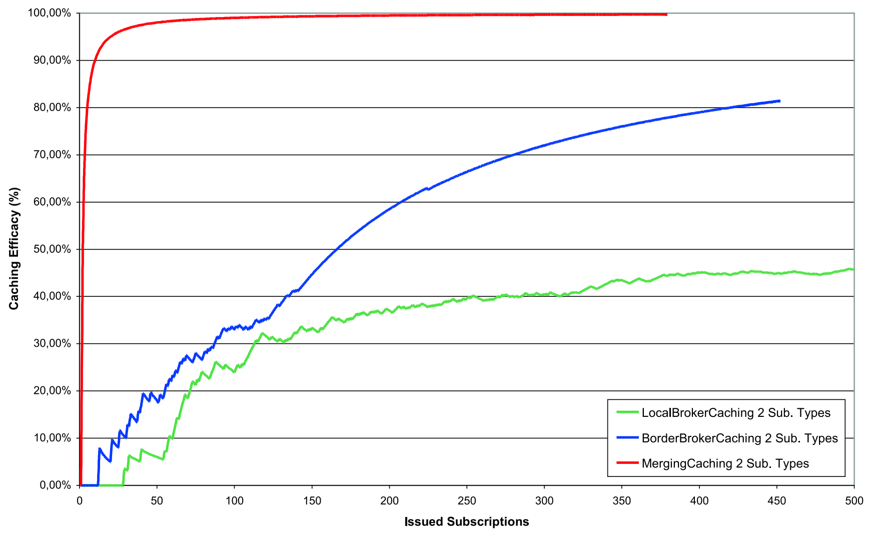
However, in order to achieve this behavior, even more processing (i.e., filter similarities checking, message interactions, creation of helper objects, etc.) has to be performed by the brokers. Given that series overlap each other in the chart, the next subsection proceeds by providing a comparison between the strategies for fixed numbers of subscription types. This arrangement will let us observe each series' caching efficacy values separately.

6.2.4 Strategies Comparison

In this subsection we look at the extracted log data from another point of view in order to provide a uniform comparison between the strategies. Here the caching efficacy values are arranged by number of subscription types (for 1, 2, 3, 4, 5 and 10 subscription types). As it is expected from the previous charts, the best efficacy is obtained by `MergingCaching`, followed by `BorderBrokerCaching` and finally `LocalBrokerCaching`.

Figure 6.5 contains three comparisons, where the upper, middle and lower charts show the caching efficacy values when 1, 2 and 3 subscription types are considered, respectively. Here, the `BorderBrokerCaching` strategy displays the most fluctuating behavior. Also the difference in the time needed to arrive to a given efficacy becomes evident: whilst `MergingCaching` rapidly grows to almost a 100%, the others require at least 100 subscriptions in order to get stabilized.





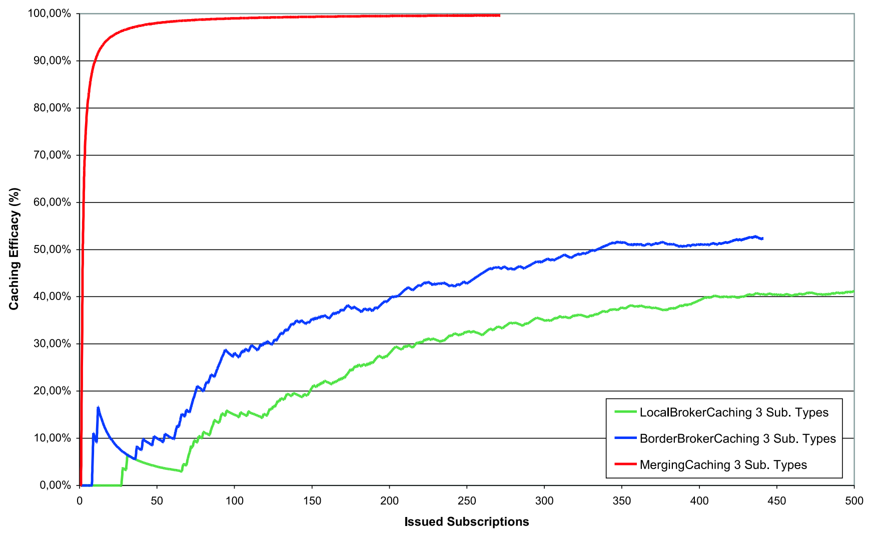
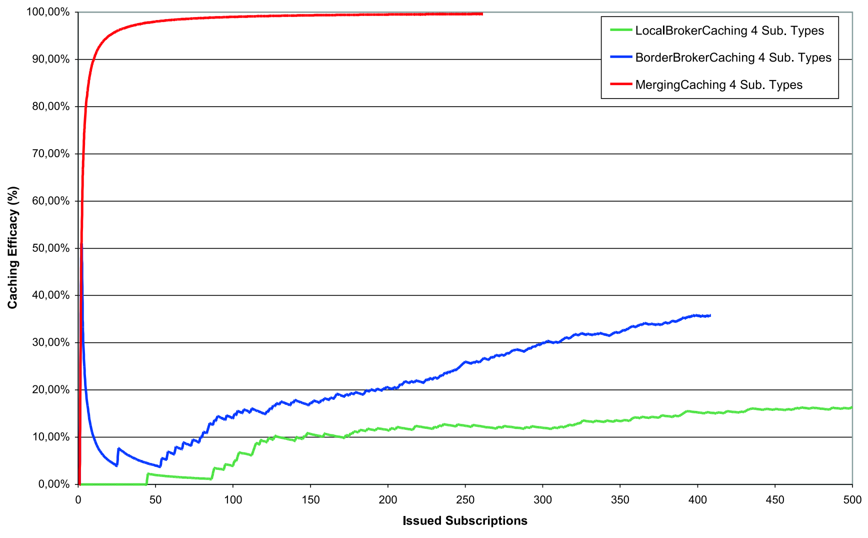
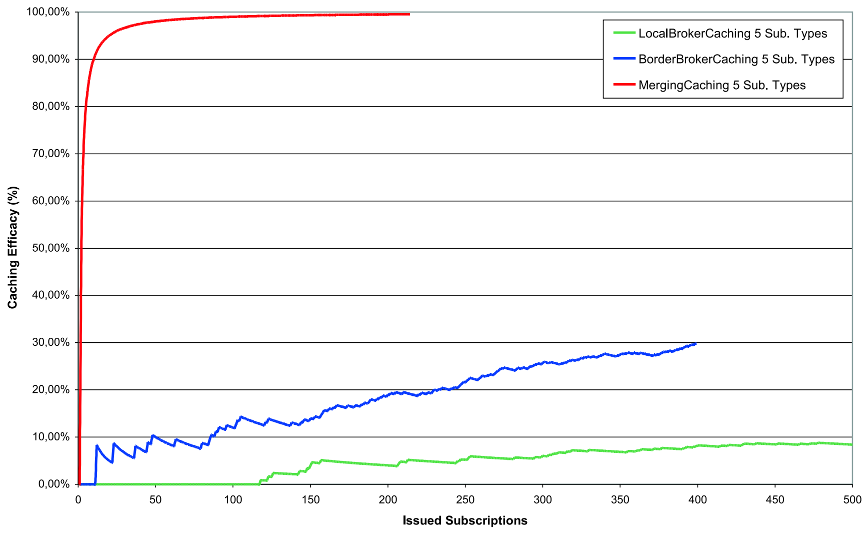


Figure 6.5: Caching Strategies Comparative Evaluation (1)





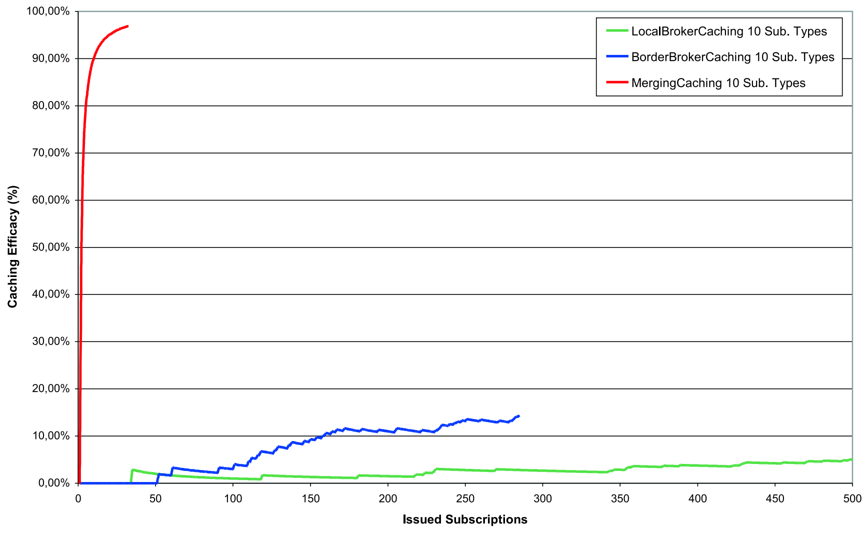


Figure 6.6: Caching Strategies Comparative Evaluation (2)

In turn, Figure 6.6 displays another three comparisons, where the upper, middle and lower charts show the caching efficacy values when 4, 5 and 10 subscription types are considered, respectively. An interesting issue that becomes clear here is the fact of the series not being complete (i.e., did not finish issuing 500 subscriptions). For instance, with `BorderBrokerCaching` only 408, 399 and 285 subscriptions were issued respectively, whereas with `MergingCaching` only 261, 214 and 32 subscriptions were issued respectively (in a timely manner). This is exclusively because of pragmatical restrictions that we describe next.

6.3 Summary

In this chapter we have showed from a different point of view how the usage of the caching strategies minimizes the bootstraping latency. In this regard, the system can be evaluated in several ways, for instance:

- Varying the quantity of past notifications required to bootstrap:

  Although the experiments carried out in the present work only required 1 notification from the past, this number can easily be changed to another arbitrary value. In such case, the logged values reflect how did the caching strategy perform (i.e., how many notifications was it able to return), in the form of fractional values in the interval [0.0, 1.0]. Hence the best-effort metric ("request fulfilled or not") is not appropriate here since it must deal with fractional values. Therefore, other metrics must be devised that allow to compare, in a uniform way, the caching strategy behavior.

- Varying the caching data structure's size:

  In the experiments we have used a combination of `MaxGlobalBuffers`, `MaxIndexedFilters` and `MaxFiltersBuffers` of 151, 15 and 10, respectively. This means that all the issued subscriptions fitted in the data structure. Nevertheless, it should be analyzed what happens when other values for these parameters are used, or even when they are variable (i.e., each broker has its own combination of values).

- Varying the locality of interest:

  The experiments were arranged with a uniform distribution of the client's interests. However, it is reasonable to state that in most mobile environments this doesn't hold. With this premise further investigations should be made that point at correlating consumer's behavior against caching strategies.

Finally, it is important to remark the conditions in which the evaluations were performed. These parameters definitively restrict the evaluation possibilities. The most important ones are the fact of carrying out the tests in a single PC (instead of a laboratory with a networked pool of computers), as well as the limited physical memory. After solving these issues, it would be interesting to perform the evaluations with a higher number of issued subscriptions (instead of 500), augmenting the number of subscription types (our greatest amount was 10), and extending the broker topology to 5 levels (instead of 4). All these factors would increase considerably the validity of current results.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This work was motivated by information-driven applications, particularly on distributed event-based systems. Developers of such systems face up a problem found at application's runtime startup. These applications require an initial phase to correctly interpret the current flow of notifications and commence normal operation, what we call *bootstrapping phase.* This stage is necessary in order to bring the application into a consistent state, and during this period an event-based application might not be able to work properly. In mobile environments this problem is aggravated. Provided that disconnections must be considered as part of normal wireless communication, and that with each reconnection the system must re-register the client's subscriptions (because of a new location, a context change, a communication failure, etc.) the problem becomes more relevant.

Considering this landscape, this work concentrates in enhancing the mobile pub/sub middleware (in particular, the notification service) by reducing the bootstrapping latency. Our approach focuses in extending the pub/sub system to store recently published notifications in caches that are distributed in the network. In this way, during the bootstrapping phase, the system is able to deliver the most recently published notifications, in order to minimize the bootstrapping latency. A client application specifies *how many (or how old)* notifications does it need to bootstrap, and the system uses this information in order to provide the required notifications. The approach we have taken is limited to best-effort, since it can not guarantee all answers because this strongly depends on the size of buffers and the traffic of messages.

We extended the system behavior by including into the notification service a component that intervenes in the client subscription process by adding the caching functionality. By observing the routing functionality, several caching mechanisms came to our minds that allowed to devise different strategies. These strategies varied, for instance, in their simplicity, effectiveness of notification fetching, memory overhead, network utilization, data access mechanisms or requirements from the infrastructure.

The enhancement was designed to be incorporated to an open source notification service. Basically, we wanted to provide the caching functionality as an add-on. That means that we did want to minimize changes on the source code of the notification service. In Figure 7.1 we describe the overall proposed architecture. Our solution is composed of three layers:

- In the lower level, the Notification Service provides the core routing functionality.
- The second level is composed of the Caching concern, which basically demarcates where a caching strategy might be applied.
- The third level is composed by a pluggable CachingStrategy that implements a defined caching behavior. These strategies can be customized, so parameters such as MaxIndexedFilters might be passed to the constructor when they are created.

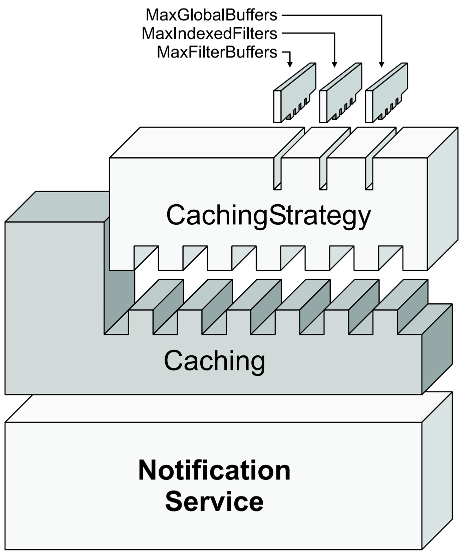


Figure 7.1: Schematic view of the enhanced notification service

Given this, we took advantage of aspect orientation tools in order to solve our problems: implement the caching strategies as a non-functional concern. For this purpose, we have used AspectJ as an adequate, seamless, aspect-oriented extension to Java. Additionally, some yet non-standardized aspect-oriented UML diagrams were used to illustrate several viewpoints of the strategies' development.

Finally, a practical evaluation of the implemented caching strategies was performed. The evaluation aimed at providing more details of the caching behavior. Many considerations must be reasoned in preparing the tests such as the broker topology, the subscribing rate or even the data model. However, an important factor emerged while carrying out the tests that was the number of subscription types. This factor divided the implemented strategies into two groups: those that depend on the number of subscription types (`LocalBrokerCaching` and `BorderBrokerCaching`) and those who don't (`MergingCaching`). The metric used in the evaluation pointed out best-effort results only, though other metrics can be used that give insights on other aspects of the behavior. For instance, when the number of required notifications from the past is variable, we could monitor fractional values that represent how well did the chosen strategy perform and analyze the outcome also in a uniform way.

7.2 Future Work

Enhancing mobile pub/sub middleware has been a major task. Each step has revealed potential areas of future work. However, time is finite and we suffice ourselves by having arrived to results at each stage.

With respect to the design, one of the requirements elicited in Section 4.1 (page 21) related to the integrability was the *dynamism*. Our solution (described in Section 5.3.2, page 63) made use of the words *startup-time* to refer the moment when the decision of deploying (or not) a given caching strategy was taken. There, we made clear that in order to incorporate the caching functionality, the compilation process must participate in order to *weave* the aspect to the rest of the system. On the other hand, if we think of a production environment, it would be nice to be able to take out the caching functionality (as well as any other add-ons) from the notification service. The basic idea is to allow the service administrator to dynamically choose to insert or remove the caching functionality *on-line*, i.e, at run-time. *Dynamic AOP* can give us a hand here.

Several mechanisms currently exist, while others are still in research, that allow the modification of an application behavior dynamically. AOP's off-line weaving is performed at application build time (like AspectJ does). Dynamic AOP's online weaving can be achieved in several ways: weaving at class load time (i.e., at application deployment time), weaving at runtime (i.e, in a running application), or even through runtime inspection and monitoring (i.e., using the Java Virtual Machine Debug Interface [3]). Finding new architectural means that allow to implement this dynamic behavior is a very rich area of future work, and it should be considered for other QoS issues on publish/subscribe notification services.

Additionally, it would be interesting to investigate how difficult it is to try to use the same caching design over other (open source) notification services. Commonly, adding an indirection level in traditional systems solves this kind of situations. Nevertheless, the layer we are talking of is mainly an *aspect*. Making the layer more generic in order to deploy it over different systems is not just finding a new set of *pointcuts*. The strategies' requirements from the infrastructure must be compared against the notification service characteristics to see if they are compatible. For example, broker topologies with cycles, which diminish single points of failure, are not supported by the developed strategies. This may influence an adaptation of the caching layer making it more generic and usable in other contexts.

The work presented in Chapter 6 can be seen as a starting point for several duties. On one hand, an initial task would be developing analytical models that allow to foresee the results to be expected and why. On the other hand, extending the tests to perform large scale simulations is necessary. In principle, the biggest problem is to have hardware available to perform the tests. This would increase the validity of current results.

Finally, and as indicated in [15], trust and security are not part of the pub/sub paradigm. Security is a separate aspect of publish/subscribe, outside of the pure ability to convey messages. To achieve greater flexibility at implementing security aspects, the employment of aspect-oriented programming techniques and AspectJ was proposed. This posses two problems: First, and as it was stated in Section 5.1.2 (page 47), when more than one aspect participates in the design of a system, it must be determined which aspect has the highest priority. A second issue is the *interaction* between the caching and security concerns. Not only a priorization must be defined to establish how the aspect composition process must weave the aspects to the components, but the interaction between them is. For example, security policies might enforce events to be delivered within its *time to live* period. Nevertheless, the events might be cached somewhere by a strategy for longer periods and later delivered to a newly subscribed client. Another example would be a caching strategy trying to extract notifications from private subnets in order to fulfill a consumer request from the past. Hence, mechanisms must be devised that allow extending the notification service while considering the interactions between other existing or incoming concerns.

Appendix A

Aspect Oriented Software Development

Programming languages evolved from machine code and assembly languages to a variety of paradigms such as formula translation, procedural programming, functional programming, logic programming, and object-oriented programming. This evolution has improved the ability to achieve a clear *separation of concerns*, or *the ability to identify, encapsulate, and manipulate only the parts of software that are relevant to a particular concept, goal, or purpose.* Nowadays, Object-Oriented Programming (OOP) has become the dominant programming paradigm where a problem is decomposed into objects that abstracts behavior and data in a single entity.

Nevertheless, although OOP technologies offers the ability for separation of concerns, it's still difficult to model and implement crosscutting concerns. This is because OOP tries to localize concerns which do not fit naturally into a single program module, or even several closely related program modules. Concerns can range from high-level notions such as *security* and *quality of service* to low-level notions like *buffering*, *caching*, and *logging.* They can also be functional, such as business logics, or non-functional, such as synchronization. Some concerns, such as XML parsing and URL pattern matching, usually couple with a few objects, yet achieve good cohesion. Other concerns, such as logging, will intertwine with many highly unrelated modules.

A.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has been proposed as a technique for improving separation of concerns in software. AOP builds on previous technologies, including procedural programming and OOP, that have already made significant improvements in software modularity. The central idea of AOP is that while the hierarchical modularity mechanisms of object-oriented languages are extremely useful, they are unable to modularize all concerns of interest in complex systems. Instead, in the implementation of any complex system, there will be concerns that inherently *crosscut* the natural modularity of the rest of the implementation. AOP does for crosscutting concerns what OOP has done for object encapsulation and inheritance: it provides language mechanisms that explicitly capture crosscutting structure. This makes it possible to develop crosscutting concerns in a modular way, and achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse. A well-modularized crosscutting concern is called an *aspect* [23].

A.2 AspectJ

AspectJ is a simple and practical Aspect-Oriented extension to Java. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns. In AspectJ's join point model the following terms are used:

- *join points* are well-defined points in the execution of a program;
- *pointcuts* are collections of join points;
- *advice* are special method-like constructs that can be attached to pointcuts; and
- *aspects* are modular units of crosscutting implementation, comprising pointcuts, advice, and ordinary Java member declarations.

AspectJ code is compiled into standard Java bytecode. Simple extensions to existing Java development environments make it possible to browse the crosscutting structure of aspects in the same way one browses the class inheritance structure. For instance, to implement the prototypical strategies of this thesis, IBM's open source Eclipse platform was used, along with a plug-in that adds the AOP functionality, called Eclipse AspectJ Development Tool. Several examples show that AspectJ is powerful, and that programs written using it are easy to understand [23]. Next we provide an overview of how an aspect is described in terms of more elementary components.

A.2.1 Joinpoints and Pointcuts

Consider the following Java class:

```
class Point {
    private int x, y;

    Point(int x, int y) { this.x = x; this.y = y; }

    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }

    int getX() { return x; }
    int getY() { return y; }
}
```

In order to get an intuitive understanding of AspectJ's joinpoints and pointcuts, we go back to some of the basic principles of Java. Consider the following method declaration in class `Point`:

```
void setX(int x) { this.x = x; }
```

This piece of program says that when method named `setX` with an `int` argument is called on an object of type `Point`, then the method body { `this.x = x;` } is executed. Similarly, the constructor of the class states that when an object of type `Point` is instantiated through a constructor with two `int` arguments, then the constructor body { `this.x = x; this.y = y;` } is executed.

One pattern that emerges from these descriptions is *"When something happens, then something gets executed"*.

In object-oriented programs, there are several kinds of *things that happen* that are determined by the language. We call these the Java joinpoints. Joinpoints consist of things like method calls, method executions, object instantiations, constructor executions, field references and handler executions. Pointcuts pick out these joinpoints.

Pointcut definitions consist of a left-hand side and a right-hand side, separated by a colon. The left-hand side consists of the pointcut name. The right-hand side consists of the pointcut itself. For example, the pointcut picks out each call to `setX(int)` or `setY(int)` when called on an instance of `Point`:

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                    call(void setY(int)));
```

As it can be seen, the pointcut is defined in terms of the composition of other (in this case *primitive*) pointcuts. The composition can be achieved by means of the logical operators *and* (spelled `&&`), *or* (spelled `||`), and *not* (spelled `!`). This allows the creation of very powerful pointcuts from the simple building blocks of primitive pointcuts.

The left-hand side definition of a pointcut can also consist of the pointcut parameters (i.e. the data available when the *events* happen). The previous sample pointcut is given the name `setters` and no parameters on the left-hand side. An empty parameter list means that none of the context from the join points is published from this pointcut. But consider another version of version of this pointcut definition:

```
pointcut setter(Point p): target(p) &&
                          (call(void setX(int)) ||
                           call(void setY(int)));
```

This version picks out exactly the same joinpoints. But in this version, the pointcut has one parameter of type `Point`. This means that any advice that uses this pointcut has access to a `Point` from each joinpoint picked out by the pointcut. Inside the pointcut definition, this `Point` named `p` is available, and according to the right-hand side of the definition, that `Point p` comes from the target of each matched joinpoint.

A.3 AOP and UML

As AOP techniques move into mainstream use, it is likely that more software developers will be modelling systems with aspect-oriented features using the Unified Modelling Language (UML). Why choose UML to visualize information? Because it provides a single, standardized, powerful language for precisely describing systems design and software design, which can be interchanged with other UML users.

While in the last years the *implementation level* has received most of the research efforts (since without programming languages to support aspects, there was no reason to consider other aspect-oriented development), very less work exists at early stages like requirements engineering, analysis and design. Nevertheless, some extensions to the classical UML notation were used in these stages in order to accommodate aspects, as proposed by the current literature:

- In order to handle the separation of crosscutting concerns at *requirements level* using UML, we can identify and specify crosscutting concerns in separate modules, so that localization and hence, reusability and maintainability can be promoted [27].
- At the *analysis and design level*, [5] proposes Aspect Packages, Class Diagrams for Aspects, Interaction Diagrams for Aspects (that is, sequence and collaboration diagrams), Statecharts with Aspects and more. At a lower level, [34] proposes the Aspect-Oriented Design Model, which allows a more precise specification of the crosscutting elements and other diagrams. In [35], the representation of joinpoints using UML is discussed. They propose the usage of UML links in sequence interaction diagrams to specify the behavioral crosscutting. We prefer the usage of simple dynamic call graph diagrams (cf. [21]), though.
- In the *design level* (but rather oriented to the implementation/coding phase), [36] propose an extension to the UML to support aspects properly without breaking the existing UML specification and an XML-based aspect description language.

Although these tools have not yet been defined as UML 2.0 standards or been approved by the OMG (at least at the moment of writing this thesis), what we expect from them is to get a better way to express the ideas applied throughout this work.

Bibliography

[1] José Antollini, Mario Antollini, Pablo Guerrero, and Mariano Cilia. Extending Rebeca to Support Concept-based Addressing. Technical report, First Argentine Symposium on Information Systems, Córdoba, Argentina, September 2004.

[2] AspectJ Team. AspectJ Programming Guide. URL: http://aspectj.org/doc/dist/progguide/index.html, September 2001.

[3] Swen Aussmann and Michael Haupt. Axon – Dynamic AOP through Runtime Inspection and Monitoring. *First Workshop on Advancing the State-of-the-Art in Run-Time Inspection (ASARTI'2003)*, April 2003.

[4] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality Attributes. Technical Report CMU-SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1995.

[5] Mark Basch and Arturo Sanchez. Incorporating Aspects into the UML. In *Third International Workshop on Aspect-Oriented Modelling (AOSD'2003)*, Boston, USA, March 2003.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 5th edition, May 1999.

[7] J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

[8] C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. An infrastructure for Meta-Auctions. In *Second International Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, San José, California, USA, June 2000.

[9] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-Based Addressing and Routing: A General Model and its Application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[10] Antonio Carzaniga and Alexander L. Wolf. Content-Based Networking: A New Communication Infrastructure. *NSF Workshop on an Infrastructure for Mobile and Wireless*, October 2001.

[11] M. Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments.* Ph.D. Thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, August 2002.

[12] Mariano Cilia, Ludger Fiege, Christian Haul, Andreas Zeidler, and Alejandro Buchmann. Looking into the past: Enhancing mobile publish/subscribe middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, San Diego, California, June 2003. ACM Press.

[13] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[14] Ludger Fiege, Gero Mühl, and Alejandro P. Buchmann. An Architectural Framework for Electronic Commerce Applications. In *Informatik 2001: Annual conference of the German Informatics Society (GI)*. ACM, 2001.

[15] Ludger Fiege, Andreas Zeidler, Alejandro Buchmann, Roger Kilian-Kehr, and Gero Mühl. Security aspects in publish/subscribe systems. In *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*, May 2004.

[16] M. Franklin and S. Zdonik. Data in Your Face: Push Technology in Perspective. In *Proceedings ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 98)*, pages 516–519, Seattle, WA, USA, June 1998. ACM Press.

[17] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 15th edition, September 1998.

[19] Kim Haase. *Java Message Service API Tutorial*. Sun Microsystems, 2002.

[20] Dennis Heimbigner. Adapting publish/subscribe middleware to achieve Gnutella-like functionality. In *Coordination Models, Languages and Applications, Special Track at 2001 ACM Symposium on Applied Computing (SAC 2001)*, pages 176–181. ACM Press, 2001.

[21] Erik Hilsdale and Gregor Kicsales. Aspect-Oriented Programming with AspectJ. Xerox Corp, July 2004.

[22] J. Kaiser and M. Mock. Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). In *Second Int'l Symposium on Object-Oriented Distributed Real-Time Computing Systems*, 1999.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[24] M. Langheinrich, F. Mattern, K. Romer, and H. Vogt. First steps towards an Event-based infrastructure for smart things. In *Ubiquitous Computing Workshop (PACT 2000)*, Philadelphia, PA, USA, October 2000.

[25] C. Liebig, B. Boesling, and A. Buchmann. A notification service for next-generation IT systems in air trafic control. In *Proceedings GI-Workshop'02: Multicast-Protokolle und Anwendungen*, Braunschweig, Germany, May 1999.

[26] C. Mascolo, W. Emmerich, and L. Capra. *Middleware for Mobile Computing*, volume 2497, pages 20–58. E. Gregori, G. Anastasi and S. Basagni, ACM Press, Springer Verlag edition, 2001.

[27] Joao Araujo Moreira, Ana Moreira, Isabel Brito, and Awais Rashid. Aspect-Oriented Requirements with UML. *Workshop on Aspect-Oriented Modelling with UML (held with UML 2002)*, 2002.

[28] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt Univ. of Technology, http://elib.tu-darmstadt.de/diss/000274/, 2002.

[29] Gero Mühl, Ludger Fiege, Felix C. Gärtner, and Alejandro P. Buchmann. Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems. In *Proc. MASCOTS 2002*, 2002.

[30] Object Management Group. CORBA Notification Service Specification. Technical report telecom/98-06-15, Object Management Group (OMG), Fammingham, MA, May 1998.

[31] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus — An Architecture for Extensible Distributed Systems. In Barbara Liskov, editor, *Proceedings of the 14th Symposium of Operating Systems Principles (SIGOPS)*, pages 58–68, Asheville, NC, USA, December 1993. ACM Press.

[32] Abraham Silberschatz and Peter Baer Galvin. *Operating Systems Concepts*. World Student Series. Addison-Wesley, John Wiley and Sons, Inc., 5th edition, 1998.

[33] N. Skarmeas and K. Clark. Content-based routing as the basis for intra-agent communication. In *Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98)*, Berlin, Germany, July 1999.

[34] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Designing Aspect-Oriented Crosscutting in UML. In *Workshop on Aspect-Oriented Modeling with UML, AOSD*, Enschede, The Netherlands, April 2002.

[35] Dominik Stein, Stefan Hanenberg, and Rainer Unland. On Representing Join Points Using UML. In *Second International Workshop on Aspect-Oriented Modeling with UML*, September 2002.

[36] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *3rd Aspect-Oriented Programming (AOP) Workshop at 13th European Conference on Object Oriented Programming (ECOOP'99)*, pages 299 – 300, Lisbon, Portugal, June 1999.