University of Zagreb
Faculty of Electrical Engineering and Computing

DISSERTATION

Service Architecture for Content Dissemination to Mobile Users

Ivana Podnar

Zagreb, February 2004

The doctoral dissertation has been completed at the Department of Telecommunications of the Faculty of Electrical Engineering and Computing, University of Zagreb.

Advisor: Ignac Lovrek, Ph.D.
Professor, FER, University of Zagreb

Co-advisor: Mehdi Jazayeri, Ph.D.
Professor, Technische Universität Wien

The dissertation has 148 pages.
Dissertation number:

The dissertation evaluation committee:

1. Branko Mikac, Ph.D., professor, FER, University of Zagreb
2. Ignac Lovrek, Ph.D., professor, FER, University of Zagreb
3. Mehdi Jazayeri, Ph.D., professor, Technische Universität Wien

The dissertation defense committee:

1. Branko Mikac, Ph.D., professor, FER, University of Zagreb
2. Ignac Lovrek, Ph.D., professor, FER, University of Zagreb
3. Mehdi Jazayeri, Ph.D., professor, Technische Universität Wien
4. Mario Žagar, Ph.D., professor, FER, University of Zagreb
5. Manfred Hauswirth, Ph.D., École Polytechnique Fédérale de Lausanne, Switzerland

Date of dissertation defense: May 7, 2004

Acknowledgments

A number of people have supported me during the process that led to the completion of the thesis. Firstly, I would like to thank my advisor Prof. Ignac Lovrek for his support, guidance, and patience throughout my research. I thank my co-advisor Prof. Mehdi Jazayeri for teaching me the principles of research, and impelling me to pursue my research interests. I am also grateful to Dr. Manfred Hauswirth who has directed my research interests towards the area of publish/subscribe systems, and supported my efforts with insightful comments and arguments.

The major ideas presented in the dissertation have been developed at the Distributed Systems Group, Technical University of Vienna. I thank the members of the DSG for creating an inspiring research environment, and for making my stay in Vienna a pleasant experience. I also thank my colleagues at the Department of Telecommunications, FER Zagreb, for their patience and understanding to reduce my teaching obligations during the last frantic months of thesis completion. Furthermore, special thanks go to FER students Krešimir Pripužić, Matija Mićin, and Branimir Turk for their participation in the implementation of the MOPS system and the m-NewsBoard application.

And finally, I cordially thank my family for their constant support, and encouragement to always do my best while pursuing my goals.

Abstract

The dissertation presents an architecture and an implementation of efficient and personalized content dissemination service targeting mobile users. The service enables information publishers to publish the content for numerous users based on the publish/subscribe interaction style. Service personalization is achieved through subscriptions: Users define subscriptions to express their interest in receiving certain content types. The published content contains non-realtime data of variable bandwidth demands (short text messages, images or video clips) and the publishing time is usually randomly determined. Furthermore, the service enables personal mobility, i.e., a user can receive the content in various networks applying different terminals. The research is motivated by the increased demand for the push-based dissemination of personalized content to mobile users that enables service users to promptly receive important notifications.

The thesis investigates two aspects of content dissemination. Firstly, a mathematical model of distributed publish/subscribe systems is presented, followed by the definition of routing algorithms that support publisher and subscriber mobility. Secondly, we propose a software architecture for content dissemination services that uses publish/subscribe middleware as its basic communication component.

The proposed model presents distributed publish/subscribe systems as discrete event systems. We propose a novel approach to routing in mobile environments that relies on notification persistency. System brokers store persistent notifications until their validity period expires, and deliver valid notifications to subscribers when they reconnect to the system. The prototype implementation and the experimental results show validity of the proposed solution. In the second part of the thesis we propose a component-based content dissemination service architecture that is adequate for mobile settings. We have designed a generic Web-based solution for the publish/subscribe component that forms the basis of the proposed architecture, and outline the solution for personal mobility. The implementation of the m-NewsBoard system, a news dissemination service for mobile users, demonstrates applicability of the proposed architecture.

Sažetak

Disertacija predlaže arhitekturu i implementaciju usluge za učinkovitu i personaliziranu isporuku sadržaja pokretnim korisnicima. Usluga omogućuje objavljivanje sadržaja na načelu objavi-pretplati namijenjenog velikom broju korisnika. Usluga je personalizirana jer korisnici pretplatom izražavaju interes za primanje određene vrste sadržaja. Sadržaj čine podaci koji se ne prenose u stvarnom vremenu, varijabilnih su prometnih karakteristika (kratke tekst poruke, slike ili video isječci), a trenutak njihovog objavljivanja je slučajni događaj. Usluga treba omogućiti pokretljivost osobe, tj. mogućnost primanja sadržaja u raznovrsnim mrežama i na različitim terminalima. Motivaciju za njen razvoj čine potrebe korisnika za uslugom koja omogućuje aktivnu isporuku personaliziranog sadržaja i time pravovremeni pristup važnim informacijama, a podržava pokretljivost korisnika.

Disertacija daje dva pogleda na uslugu za isporuku sadržaja. Najprije je predložen matematički model koji opisuje distribuirane sustave objavi-pretplati, te su definirani algoritmi umjeravanja poruka koji podržavaju pokretljivost korisnika sustava. Potom je predložena arhitektura usluge za isporuku sadržaja temeljena na sustavu objavi-pretplati koji čini osnovnu komunikacijsku komponentu usluge.

Predloženi model opisuje distribuirane sustave objavi-pretplati kao sustave vođene diskretnim događajima. Predložen je novi pristup usmjeravanju poruka u pokretnoj okolini temeljen na perzistentnosti poruka. Poslužitelji sustava objavi-pretplati čuvaju perzistentne poruke dok ne istekne period njihove valjanosti, te ih isporučuju pretplatnicima prilikom ponovnog spajanja u sustav. Implementacija prototipa i eksperimentalni rezultati pokazuju primjenjivost predloženog rješenja. U drugom dijelu disertacije predložena ja komponentna arhitektura sustava za isporuku sadržaja pokretnim korisnicima. Oblikovana je komponenta objavi-pretplati primjenom tehnologije Web service *kojačini osnovu arhitekture sustava, te opisan prijedlog rješenja za osobnu pokretljivost korisnika. Predložena arhitektura je verificirana implementacijom sustava* m-NewsBoard *koji se koristi za isporuku vijesti pokretnim korisnicima.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The extensive headway of mobile networks and their expected convergence with the Internet has been driven by the demand for higher rate wireless connectivity in support of terminal mobility, and the need for novel services that can satisfy the requirements of mobile users. The development of the network infrastructure with increased bandwidth and coverage has been the primary objective of network operators [55]. The next challenge is the deployment of a variety of new services that will become a necessity for mobile users, and utilize the provided bandwidth.

Personalized and flexible content dissemination has been recognized as a valuable service in the environments that support user mobility [30, 62, 94]. Notification services for weather and traffic reports, messaging systems for group discussions, and location-based information delivery services are examples of applications that rely on content dissemination. Such services enable users to receive relevant information while being mobile, and to define the type and customize the content delivered to their terminals. The main characteristic of content publication is the event-driven and probabilistic nature of content availability that cannot be predicted a priori.

Motivation. Information and notification services for communicating time-sensitive data have proved their usability in the Internet domain [24, 69, 57]. The huge success of Short Message Service (SMS) and the increased acceptance of Multimedia Message Service (MMS) advocate the extension of the initial application domain to mobile environments, and encourage further efforts to implement and deploy content dissemination services in mobile environments. However, mobile scenarios introduce additional requirements regarding the service: Mobile users want to be served with relevant and personalized content in a timely manner. Moreover, the content must be customized to their current presence status, and directed to the terminal they are currently applying. Therefore, service flexibility and its ability to deliver personalized content that provokes no nuisance to end users is of major importance for the wide acceptance of the service. Support for personal mobility is needed to assure timely information dissemination in accordance with the user's presence status.

Publish/subscribe content dissemination. Content dissemination service enables delivery of content from information sources to numerous users across a wide area network. The content being

distributed is non-realtime multimedia content such as plain text, images, or video clips. The service involves two types of users: publishers and subscribers. Publishers define and structure the content that is submitted to the service for subsequent delivery to subscribers. Subscribers define subscriptions that describe the type of content they are interested in receiving. Content dissemination service can, therefore, be visualized as an information bus that joins a publisher when publishing the content with a group of subscribers interested in the published content. The publish/subscribe communication middleware [42, 100] reflects the interaction style of content dissemination services. Communicating parties, publishers and subscribers, interact asynchronously by generating and consuming notifications. The notifications are delivered to subscribers in the push-style. The available publish/subscribe implementations [33, 24, 43, 90, 78] offer mechanisms for defining expressive subscriptions, and enable content filtering according to the provided subscriptions.

Beneficial characteristics of publish/subscribe. Publish/subscribe interaction decouples the communicating participants because they interact through an intermediary and need not to be active simultaneously to exchange messages. The decoupled, asynchronous, and persistent publish/subscribe communication is adequate for highly dynamic mobile environments: A mobile network is faced with a changing number of terminals that are often disconnected or unavailable, whereas a terminal needs to adapt to connectivity and services offered by the network in which it currently resides. Furthermore, the publish/subscribe model supports system extendibility. The addition of a new publisher or a subscriber does not affect system functionality which is desirable when dealing with frequent changes in the mobile environment. The problem of system scalability is solved by designing a distributed architecture that comprises a network of special servers, denoted as brokers in terms of publish/subscribe systems. Lastly, publish/subscribe systems satisfy the requirement for context-awareness. The event-based nature of the publish/subscribe interaction model offers the means for designing responsive applications that are aware of publisher's and subscriber's states, and adaptable to changes in the network context. Consequently, we have decided to build a mobile content dissemination service using the publish/subscribe middleware. However, the available publish/subscribe systems offer limited or no support for publisher and subscriber mobility.

Shortcomings of current approaches. SMS- and MMS-based notification services from the telecommunications domain are adjusted to mobile environments and offer push-based delivery of content to mobile terminals. However, these services lack flexibility and expressiveness of subscriptions, as well as content filtering found in publish/subscribe systems, and provide no support for content customization according to the user's presence status. On the other hand, notification services from the Internet domain offer expressive and flexible subscription mechanisms for building highly personalized content dissemination service, but offer limited support for user mobility.

We argue in [94] that the publish/subscribe middleware itself must offer mobility support and ensure seamless client reconnections preserving notifications published during disconnection. The authors in [125] agree that mobility-related issues should be addressed by the publish/subscribe middleware itself, rather than being delegated to the application layer. However, the existing pub-

lish/subscribe systems are designed and optimized for static environments where both publishers and subscribers are stationary. The problems related to mobility have been addressed recently [21, 48, 125]. The proposed solutions for client mobility in publish/subscribe middleware are not optimized for mobile environments, but they extend the existing stationary systems [48, 21]. The common solution is based on the "*queuing approach*" where a system broker – usually the last broker that served a subscriber, or a special subscriber proxy – acts as a proxy subscriber during subscriber disconnection. This proxy stores the notifications published during the subscriber disconnection in a special queue, and delivers them to the subscriber upon reconnection. If the subscriber reconnects to the system through a new system broker, a costly handover procedure is performed which transfers the stored notifications to the subscriber and updates delivery paths in the broker network.

Contributions. The thesis focuses on two aspects of content dissemination. Firstly, we design a solution for the mobility-enabled publish/subscribe middleware by defining a mathematical system model and the corresponding routing algorithms. Secondly, we propose a software architecture for content dissemination services that uses the publish/subscribe middleware as its basic communication component.

The thesis proposes a mathematical model for distributed publish/subscribe systems. The model describes publish/subscribe systems as discrete event systems that change the system state through a sequence of events. The model is defined using the set theory notation following the approach in [27]. It is the basis for defining the routing algorithms for delivery of published notifications to subscribers that are mobile, and potentially disconnected from the system.

We propose a novel approach to mobility in publish/subscribe systems. Rather than queuing notifications for disconnected users in special queues, the network of brokers stores persistent notifications until their validity period expires. We argue that notification publishers need to define validity period of published notifications, the system must assure notification storage during the validity period, and deliver valid notifications to subscribers when they reconnect to the publish/subscribe system. If a subscriber connects to the system after notification expiry, the notification is not delivered to the subscriber because it is reasonable to believe that it no longer holds valuable information. Notification persistency is by no means a new characteristic of the existing publish/subscribe systems. For example, JMS [108] defines notification persistency as one of the basic notification characteristics. However, the existing approaches do not consider notification persistency as a possible solution to the mobility problem in publish/subscribe systems.

To validate the proposed model and the routing solution we have implemented a prototype system distinguishable from other publish/subscribe implementations by its inherent support for publisher and subscriber mobility. Furthermore, we applied the prototype to evaluate the performance of the proposed routing solution, and to compare our approach with the queue-based approach.

In the second part of the thesis we have designed a component-based content dissemination service architecture. The architecture is composed of the components that have been identified through the analysis of service usage scenarios following software engineering principles [54]. We have identified

publish/subscribe and personal mobility as vital architectural components. We propose the design of a publish/subscribe component as a Web-based service, and outline a solution for personal mobility with the focus on communication patterns and security.

Finally, we present m-NewsBoard, a news dissemination service for mobile users. The m-NewsBoard system enables users to publish and receive news of their interest, and to customize the service regarding the means for news receipt. It has served as a proof of concept implementation to evaluate the proposed content dissemination service architecture.

Thesis structure. The thesis is structured as follows: Chapter 2 explains the basic concepts of content dissemination and mobility, and justifies the decision to apply publish/subscribe middleware for the design and implementation of personalized content dissemination services supporting mobile users. Chapter 3 gives an overview of the existing solutions and systems related to content dissemination. We analyze and compare the characteristics of the prominent publish/subscribe systems, and of the related solutions, such as e-mail, Usenet news, SMS, MMS, application-level multicast, and push systems. Chapter 4 introduces a mathematical model of distributed publish/subscribe systems used as the basis for defining the routing algorithms for selective dissemination of notifications to mobile subscribers. The proposed routing algorithms are defined and evaluated in Chapter 5. In Chapter 6 we propose the reference architecture for content dissemination services that comprises a Web-based publish/subscribe middleware layer, and a component for personal mobility. The implementation of a news dissemination service, m-NewsBoard, that is used to show the applicability of the proposed reference architecture is presented in Chapter 7. Chapter 8 evaluates the thesis contributions and gives guidelines for future work.

Chapter 2

Content Dissemination and Mobility

This chapter analyzes the applicability of publish/subscribe middleware for design and implementation of content dissemination services supporting user mobility. The publish/subscribe interaction model reflects the content dissemination interaction style. Content sources are publishers that provide the content to a content dissemination service which, in turn, distributes it to interested subscribers, content destinations. Subscribers can personalize the received content through subscriptions by describing the type and properties of notifications of their interest. The inherent characteristics of publish/subscribe middleware, such as loose coupling between the communicating parties and system extendibility, are suitable for the dynamic and volatile nature of mobile environments.

The chapter is organized as follows: Section 2.1 defines the content dissemination service and describes the basic interaction pattern between service entities. Section 2.2 analyzes the existing interaction models for distributed applications and classifies them across two dimensions, communication and cooperation. Section 2.3 gives a comprehensive analysis of the publish/subscribe interaction style: It describes the communication pattern, defines the notion of notification and subscription, describes the existing subscription schemes, and gives an overview of publish/subscribe characteristics. Section 2.4 defines the basic terms related to mobility and mobile networks. It presents the mobile environment for the deployment of mobile content dissemination services. Mobility management is essential for the implementation of services that support personal mobility. Section 2.4.2 gives an overview of the existing solutions to mobility management, and discusses the requirements of a mobility management solution that is adequate for content dissemination.

2.1 Introduction

The traditional request/reply model of distributed computation supports a demand-driven interaction style between two communicating parties: A client sends a request to a server that performs the computation, or hosts the requested information, and waits for a reply from the server. Information-intensive applications for timely dissemination of content to a multitude of clients face a significant performance deficiency if implemented using the request/reply approach. Clients that depend on time-

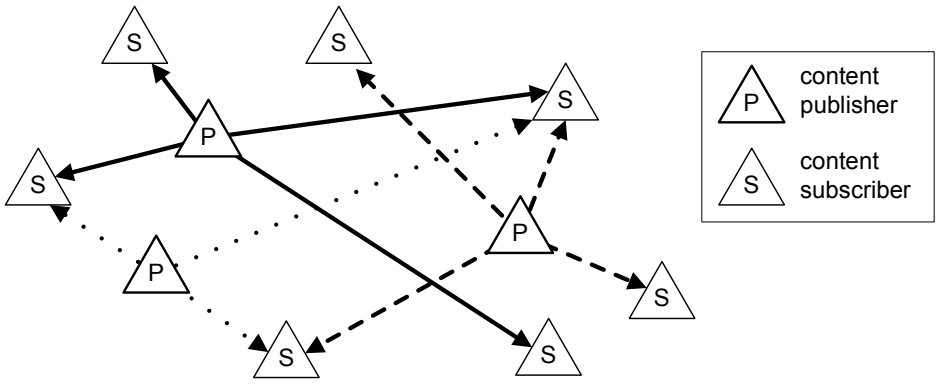

Figure 2.1: Content dissemination

sensitive information need to poll for data at regular intervals to check if the data has been updated. Inefficiency is the main drawback of this approach: The sequence of requests and replies wastes a lot of bandwidth which is particularly inadequate for mobile environments with limited wireless bandwidth and power supply. Furthermore, servers offering time-sensitive information, e.g., sport news or election results, are saturated with a multitude of requests [51].

The *push-based approach* which facilitates active dissemination of published information to interested users has been proposed as a viable solution to the problem of timely and scalable content dissemination. A *content dissemination service* enables the delivery of content from content sources to a potentially large number of interested users, content destinations. Some authors use the term *notification service* [23] to denote that the content, in most cases, carries time-sensitive information. Others use the term *push service* [57, 69] to indicate that the content is actively delivered to content destinations, as opposed to the traditional user-initiated pull-style model. Timely content dissemination is the main service task. The publishing time and the time of content change is a probabilistic event that cannot be predicted a priori. The content being distributed is non-real time multimedia content, e.g., plain text, images, or video clips.

Figure 2.1 shows the basic interaction between content sources and content destinations on the top service level. A content source publishes the content at a particular time t, and the published data is delivered to destinations having declared interest in receiving it within a "reasonably" short time interval Δt. Various scenarios for determining the set of destinations for each source and its publication are possible. For example, in a *single-source scenario* each source can have a stationary group of destinations that want to receive all of the content published by this particular source. In a *multiple-source scenario* a number of sources can publish the content to a predefined group or receivers. Publishers and receivers can either join or leave a predefined group which makes the subscription scheme rather restricted and static. Group communication services [39] including the network-layer IP multicast [35] are practical implementations of such static content dissemination services.

A dynamic and flexible subscription scenario is possible in which a group of receivers is formed

per each publication. This approach incorporates the *publish/subscribe* interaction model [100]: Subscribers declare the interest in certain content categories and notifications with particular properties, and receive only publications that match their subscriptions. A dynamic scenario facilitates personalized content delivery guided by user subscriptions. The publish/subscribe interaction enables the customization of a set of received notifications minimizing thus the amount of received content irrelevant to a subscriber. However, the gain in service flexibility and personalization complicates the design and implementation of a content dissemination service: The service must provide the mechanisms for describing the content that is of interest to subscribers, and implement the mechanisms for delivering messages matching the descriptions.

The publish/subscribe model seems a natural choice for the design of a content dissemination service since it reflects the interaction style for content dissemination. Recent solutions and measurements show that the publish/subscribe infrastructure can substantially improve performance and scalability of the traditional request/reply content dissemination services [15, 87]. Furthermore, the loosely-coupled, asynchronous, and persistent publish/subscribe model is recognized as a suitable paradigm for mobile applications requiring adaptation to highly interactive and changing conditions of mobile environments [30]. In addition, Short Message Service (SMS) and Multimedia Message Service (MMS), the prominent services from the telecommunications domain, are inherently push-based: SMS delivers text messages directly to mobile phones, while MMS can first push the notification about the receipt of a new multimedia message, and let a user request the message in the second step.

2.2 Classification of Distributed Interaction Models

Distributed systems comprise software components that run on different hosts in local or wide-area networks: Distributed processes run concurrently in different operating systems virtual processors which leads to the need for process communication and coordination. Sockets, remote procedure call and its successor, remote object invocation, message-queuing, shared dataspaces, and publish/subscribe are notable models and infrastructures used in distributed system implementations. They enable data exchange between distributed processes and offer higher level operations to the application programmer than the low-level message passing offered by the underlying transport network. The models listed have different characteristics with respect to the underlying communication and cooperation style, and offer different abstraction levels of functionality. For example, sockets are built on top of the message-oriented model of the transport layer and act as an interface to transport services provided by TCP and UDP [68]. Publish/subscribe middleware, on the other hand, offers higher level of abstraction through its special interaction model. It can be built using other lower-level models for process communication, e.g., datagram sockets, stream sockets, or even remote method invocation.

There are a number of classifications of the interaction models for distributed systems. Reference [51] classifies dissemination systems according to the following data delivery mechanisms: push vs. pull, periodic vs. aperiodic, and point-to-point vs. multi-point. The authors in [42] compare the

interaction models with respect to *space*, *time* and *synchronization* decoupling. Reference [112] focuses on messaging models, i.e., on message-queuing and publish/subscribe messaging, and presents a classification model for messaging middleware with three different models: *message delivery model*, *message processing model* and *message failure model*. The authors in [100] propose a framework for the publish/subscribe communication that comprises seven models: object model, event model, naming model, observation model, time model, notification model, and resource model.

We propose a complementary classification of distributed interaction models based on the approach presented in [115] that analyzes the characteristics of the existing models with respect to communication, interacting processes, and their coordination. We classify the models across two dimensions: *communication* and *cooperation*. Communication enables the transport of information between the interacting parties: A communication pattern defines the rules for message generation and the sequence of messages between distributed processes. Cooperation deals with the joint operation of distributed processes observed as a whole.

Communication. We classify the models according to the following communication patterns: *request/reply*, *put/get*, and *subscribe/publish/notify*. The listed communication patterns have different characteristics with respect to temporal and referential coupling, communication persistency, communication initiation, and receiver multiplicity. Distributed processes are *temporally coupled* if they need to execute simultaneously to exchange the data. *Referential coupling* denotes that a process initiating the communication knows the globally unique identifier of a destination process. *Persistent communication* ensures that the submitted information is delivered to the receiver. In contrast, *transient communication* offers no guarantees regarding information receipt and provides best-effort delivery. There are two approaches to communication initiation: pull and push. Two processes are communicating in *pull-style* when a client process requesting information sends an explicit request followed by a reply from the server process. With *push-style* communication a process registers a handler and passively waits for the incoming data. With respect to receiver multiplicity the communication can be either *point-to-point* with two interacting parties, or *multi-point* with multiple data recipients.

Request/reply. Request/reply is a simple and widely used communication pattern based on the exchange of requests followed by replies. A client process issues a request and delivers it to a server for processing. Then a server sends a reply to the client. The client is blocked while waiting for the reply or an acknowledgment of request receipt from the server. The client and server process are *temporally coupled* since they need to execute simultaneously to carry out the communication. They are *referentially coupled* because the client sends a request to the known server with the identifier of the serving process. Request/reply is the representative of *pull-style point-to-point* communication: The communication is initiated by an explicit request for information. The World Wide Web, one of the widely-used applications in todays networks, uses the request/reply interaction principle as its basic communication model.

Put/get. The main rationale behind the put/get communication pattern is *communication persistency*: The information source – sender – puts the information in a well-known storage space, and

the information destinations – receivers – get the information from the storage. Senders and receivers are *temporally uncoupled* because the operation of storing a message is independent from its retrieval. Put/get is the representative of a loosely-coupled mediated communication pattern: Senders and receivers do not interact directly, but rather through an intermediary, the data storage. The communication is performed in a pull-style because a receiver sends an explicit request for a message from the storage.

There are two representative models that fall into the put/get category: message-queuing and shared dataspaces. Message-queuing implements a point-to-point referentially coupled communication pattern in which senders put messages into receiver's queues. Shared dataspaces use a common data structure, an intermediary through which the processes exchange messages. A process puts a message into a shared dataspace, and can retrieve it from that medium. Shared dataspaces implement a one-to-many communication pattern and offer referentially uncoupled communication because the destination processes are anonymous to a process that is the information source.

Subscribe/publish/notify. Subscribe/publish/notify is a *push-style multi-point* communication pattern where the receivers of information – subscribers – first *subscribe* to a category of information and register handlers that will receive the information when published by information sources. Next, when a publisher *publishes* the content, a *notify* operation is invoked on the subscriber's side assuming that its subscription matches the published content. Publishers and subscribers can interact directly. In that case they are temporally and referentially coupled. However, most systems offer mediated solutions and use an intermediary for enabling referentially and temporally uncoupled communication. The publish/subscribe infrastructure implements the subscribe/publish/notify interaction model.

Cooperation. Cooperation enables distributed processes to act jointly in order to provide a common service. With respect to cooperation we classify the models as *address-based* and *content-based* to define the basic mode of interaction between distributed processes.

Address-based cooperation. The traditional approach to cooperation relies on the availability of the address of a communicating party and is closely related to referential coupling. The interacting parties can cooperate and communicate if they know the address of the parties they want to interact with. The address-based cooperation relies on the traditional unicast and multicast routing where messages are given explicit destination addresses that enable the transport of data to defined destinations. For example, request-reply uses direct point-to-point address-based cooperation between a client initiating a request to a well-known server. Group communication is another example of the address-based cooperation: A group is a set of processes, group members, and each group is associated with a logical name [27]. A logical name represents a group address. IP multicast [35] at the network level and application-layer multicast solutions [14]

Table 2.1: Comparison of communication and cooperation patterns

	temporal coupling	**referential coupling**	**persistency**	**communication initiation**	**receiver multiplicity**
request/reply	coupled	coupled	transient	pull	point-to-point
address-based put/get	uncoupled	coupled	persistent	pull	point-to-point
content-based put/get	uncoupled	uncoupled	persistent	pull	multi-point
address-based subscribe/publish/notify	coupled	coupled	transient	push	multi-point
content-based subscribe/publish/notify	uncoupled	uncoupled	persistent	push	multi-point

are practical implementations of group communication. Message-queuing is also an example of the address-based cooperation where each receiver has an associated addressable queue.

Content-based cooperation. The content-based cooperation is driven by the content of messages communicated among the interacting parties. Content destinations define the characteristics or templates of the content they want to receive, and the messages matching their descriptions are sent to them. This cooperation style is usually mediated and requires an intermediary that performs matching. It enables anonymous communication since the content sources do not necessarily know message recipients, and can remain anonymous to content destinations. The content-based cooperation is more flexible than the address-based cooperation because the binding between communicating parties is dynamic and data-dependent, as opposed to static binding to fixed names or addresses. Shared dataspaces and publish/subscribe are the existing models that support the content-based process cooperation.

Table 2.1 compares the defined communication and cooperation patterns. Request/reply is an address-based highly-coupled communication pattern between two processes that execute simultaneously and cannot rely on the content-based cooperation. Conversely, put/get and subscribe/publish/notify that use the content-based cooperation offer a loosely-coupled process interaction. The address-based variants are referentially coupled, and subscribe/publish/notify is also temporally coupled.

Next we give an overview of the models and infrastructures currently used in practice for the development of distributed applications.

RPC and RMI. Remote procedure call (RPC) [16] and its successor, remote method invocation (RMI), extend the principle of local method invocation into a distributed context and make the remote method call transparent to the invoking process. The main design goal is location transparency which hides the distribution of processes from the application programmer and simplifies distributed programming. The invocation of a remote method call comprises the marshaling of procedure parameters on the client side, the transport of parameters to the server side, the unmarshalling of parameters and the execution of the remote procedure on the server side. The results of procedure execution will

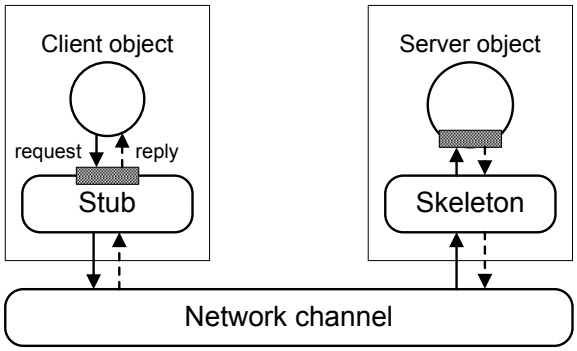

Figure 2.2: Remote method invocation

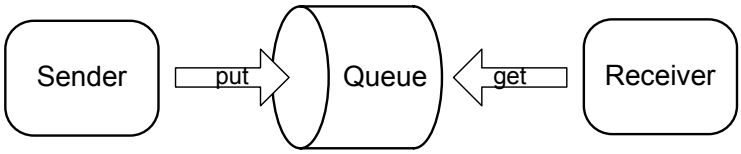

Figure 2.3: Message-queuing

be returned to the client process following the same procedure of results marshalling, transmission and unmarshalling. RMI uses the same principles adjusted to object-oriented contexts. In object-oriented systems objects interact by invoking the object methods using the interfaces that define those methods. Figure 2.2 depicts a client object that invokes a method on a remote object through a stub and skeleton that perform parameter marshaling and unmarshalling. The client-side stub holds an interface that is equal to the interface of the remote server object: From the client's viewpoint a call to the remote method is equal to a local method call.

RPC and RMI follow the request/reply communication and the address-based cooperation pattern. A client invokes a remote procedure – a method on the remote object – in a synchronous mode and blocks until the remote invocation returns. In this way, the client and the server are tightly coupled, both temporally and referentially, and the communication is transient and synchronous.

Message-queuing. The interaction style based on message-queuing follows the put/get address-based pattern that supports persistent asynchronous point-to-point communication. Distributed processes interact by putting messages into queues and by getting messages from the queues as shown in Figure 2.3. Senders and receivers are referentially coupled since a sender needs to know the identifier of the receiver's queue. Senders and receivers are temporally uncoupled because a message placed into the queue remains stored until removed by a receiver. Therefore, there is no need for the sender and the receiver to execute simultaneously.

Shared dataspaces. Shared dataspaces implement the put/get content-based interaction style. Synchronization and communication between distributed processes is performed through operations

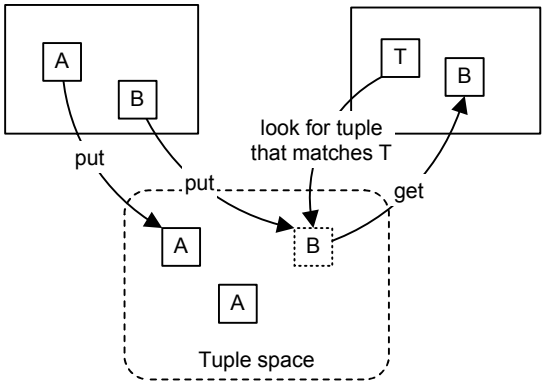

Figure 2.4: Process interaction through shared dataspace

Table 2.2: Classification of the existing models

	address-based	**content-based**
request/reply	RPC/RMI	-
put/get	message-queuing	shared dataspaces
subscribe/publish/notify	group communication	publish/subscribe

on the shared data. The shared dataspace is implemented as a distributed shared memory that stores a set of ordered *tuples*. A tuple is stored into the dataspace using the operation `put` that creates a tuple instance in the shared memory. A process that wants to read the data from the shared dataspace defines a template for matching tuple instances and uses the operation `get` to import the tuple from the dataspace as depicted in Figure 2.4. If a tuple instance matching the template is found in the dataspace, it is returned to the requesting process and optionally removed from the dataspace. JavaSpaces [109], a service used in Jini, is an example implementation of shared dataspaces. There is no distinction between clients and servers in tuple spaces. There are processes putting tuples into the shared memory and processes extracting tuples from it. The communication is potentially multi-point because a process can read a tuple without removing it from the dataspace and, thus, enable other processes to read it afterward until one of the processes removes it from the dataspace.

Publish/subscribe. Publish/subscribe systems implement the content-based subscribe/publish/notify interaction style and provide loose-coupling between the interacting parties. Notification publishers and subscribers are temporally and referentially uncoupled. The communication can be persistent because the intermediary can store notifications until they are delivered to all subscribers. One of the features that distinguishes publish/subscribe from other interaction styles is its inherent multi-point communication style. A published notification is replicated and delivered to all interested parties in the push-based style.

Table 2.2 classifies the existing models with respect to the presented communication and coordi-

nation interaction styles.

2.3 The Concepts of Publish/Subscribe

The publish/subscribe interaction model enables asynchronous communication between information *publishers* and *subscribers*. Publishers and subscribers communicate by exchanging *notifications*, often denoted as *events*, that represent information items and carry the published content. The model is *event-driven* because the act of publishing is aperiodic and guided by the availability of a new or modified information item, or by a publisher's state change. Publishers produce the information and subsequently publish it for dissemination to interested subscribers: Publishers are notification producers, while subscribers act as notification consumers that declare interest in receiving specific categories of notification. When a notification is published, it is delivered to all the subscribers that have declared interest in receiving such notification. Publishers and subscribers may interact directly: However, most systems introduce an intermediary, an "information bus" [85] responsible for efficient notification delivery from publishers to subscribers. The intermediary ensures anonymity of communicating parties: Publishers and subscribers do not necessarily know of each other and the infrastructure keeps track of their subscriptions and publications. Furthermore, the interaction style enables one-to-many multicast-style communication because the published notification is delivered to all interested subscribers.

The systems that implement the publish/subscribe interaction style fall into the category of *middleware*, software infrastructure built on top of the network operating system that offers generic services for the development of distributed applications. The main purpose of middleware application in practice is to simplify the implementation of distributed systems [40]. Publish/subscribe systems are often classified as *event-based middleware* because of the event-driven communication and cooperation model that uses notifications for carrying the information passed among the communicating parties [74, 100]. The authors in [115] classify publish/subscribe systems as *coordination systems* to stress that the publish/subscribe interaction coordinates the activities between distributed processes.

2.3.1 Publish/Subscribe Interaction Model

The publish/subscribe model involves two types of entities: publishers and subscribers. *Publishers* are content sources that publish notifications and *subscribes* are content destinations that subscribe to a number of notification types using the *publish/subscribe service*. The publish/subscribe service provides the management of subscriptions, and storage and dissemination of published notifications. It is an intermediary between the referentially uncoupled and anonymous publishers and subscribers. The interaction between publishers and subscribers is achieved through the mechanism implemented by the publish/subscribe service that matches subscribers' subscriptions with published notifications, and delivers the matching notifications to the interested subscribers.

Notifications are information items that are produced and published aperiodically. They usually

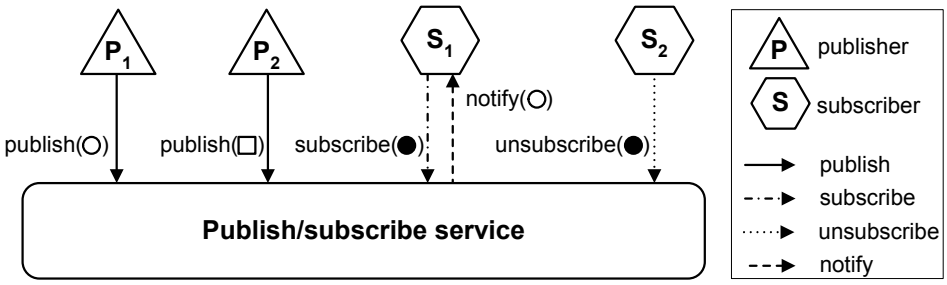

Figure 2.5: The basic publish/subscribe interaction model

contain a set of properties in the form of attribute-value pairs, and a payload. Notification properties describe the published notification and may be generic – e.g., unique identifier, timestamp, expiry field, priority – or application specific, i.e., defined by the notification producer. Notification payload carries the actual published content which can be a document of a Multipurpose Internet Mail Extension (MIME) type [52]. To transmit a notification across the network, an entity, either a publisher or an intermediary service, must incorporate the notification into a *message*. Messages are exchanged between two parties, a source and a destination. They are generally composed of a header field, that contains the source and destination addresses, and a payload field, which contains the transported notification.

Publishers and subscribers interact according to the subscribe/publish/notify communication pattern. Figure 2.5 shows the basic operations that enable the interaction between publishers and subscribers through an intermediary publish/subscribe service. A subscriber expresses its interest in receiving notifications of a certain type by invoking the method `subscribe` on the publish/subscribe service. The method `unsubscribe` is used for subscription termination. For example, in Figure 2.5 subscriber S_1 subscribes to "circular" notifications by invoking the method `subscribe` of the publish/subscribe service. S_1's subscription is defined by a property that is common to the notifications of interest: S_1 is interested in "circular" notifications. A subscription can be viewed as a notification description or a template: The publish/subscribe service stores subscriptions and uses them for comparison with the published notifications.

When generating a notification, a publisher invokes the method `publish` of the publish/subscribe service and supplies the notification. The notification contains some information or content that a publisher has and wants to share with other interested parties. For example, in Figure 2.5 the publisher P_1 publishes the notification ◯ and the publisher P_2 publishes the notification □. Next, the publish/subscribe service compares each published notification with the existing subscriptions, and eventually delivers ◯ to the interested subscriber S_1 using the method `notify`. The same notification is not delivered to S_2 because S_2 has terminated the subscription to "circular" notifications before ◯ notification is published. If a subscriber cannot be contacted, a published notification is discarded, or stored to be delivered when the subscriber becomes available, depending on the persistence of the published notification.

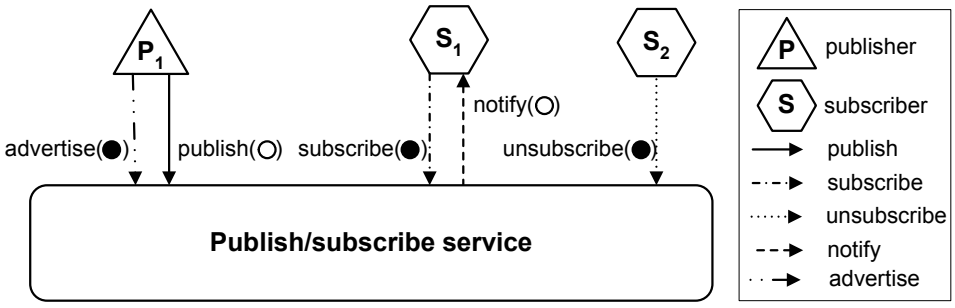

Figure 2.6: The extended publish/subscribe interaction model

The methods `subscribe`, `unsubscribe`, `publish`, and `notify` comprise the methods that are used in the basic publish/subscribe interaction model. The extended model incorporates the method `advertise` which allows a publisher to announce the intent of publishing notifications of a defined type. Publishers can terminate their advertisements by invoking the method `unadvertise`. Figure 2.6 depicts a scenario in which the publisher P_1 first advertises future publications of "circular" notifications. Next, the subscriber S_1 defines a subscription to "circular" notifications. Subsequently, when P_1 publishes the notification ◯, the publish/subscribe service delivers it to S_1. Advertisements are used to define notification types that will be used in the publish/subscribe system. Additionally, advertisements enhance the efficiency of the routing protocols in distributed architectures of publish/subscribe services.

2.3.2 Subscription Schemes

The publish/subscribe interaction model enables subscribers to specify the categories of notifications they want to receive: Subscribers are usually interested in a subset of published notifications, and subscriptions are used to filter out the published notifications according to individual subscriber's needs. Subscriptions can be regarded as notification templates: They contain the rules that enable the publish/subscribe service to compare each published notification with the subscriber's interest, and to deliver only the notifications that match the defined subscription criteria. The publish/subscribe service can thus be viewed as a notification filter that performs the process of matching notifications to subscriptions. Currently, the publish/subscribe systems use three different schemes for subscriptions: *subject-based*, *content-based*, and *type-based* subscriptions.

Subject-based subscription. Subject-based, or topic-based subscriptions classify each notification as belonging to a particular subject, i.e., topic. A subject is used to characterize and classify the published content and can be regarded as a logical connector between publishers and subscribers. Subjects can be arranged in a hierarchy: A sub-subject can be derived from a super-subject to further specialize the notifications published on the super-subject.

Subject-based subscriptions are an extension of the *channel-based* subscription model used in

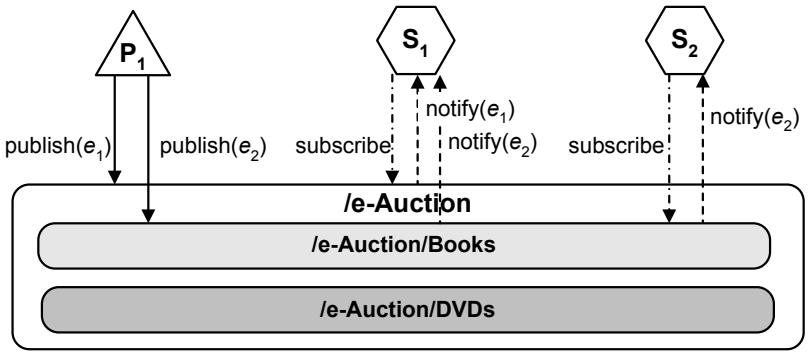

Figure 2.7: Subject-based subscription scheme

the first publish/subscribe systems, e.g. CORBA event service [82]. Channels are similar to the notion of groups defined in the context of group communication [42]: When a subscriber S_i subscribes to a channel ch_j, S_i becomes a member of the group of subscribers to the channel ch_j. A notification published on channel ch_j is delivered to all subscribers of the channel. Channels can be implemented efficiently using IP multicast because each channel can be mapped to a multicast group: The Global Information Broadcast is a push system for information distribution that uses IP multicast or reliable multicast for the efficient data transport at the network level [69]. However, the main disadvantage of the channel-based notification classification is its limited expressiveness with a coarse level of notification classification.

The subject-based approach uses *subject hierarchies* to offer finer granularity of notification categories. Subjects are usually specified using the URL-type format, e.g. `/e-Auction/Books/ScienceFiction`. Subscribers can subscribe to a particular subject, or refine their subscription by subscribing to a sub-subject. It is even possible to use a more complex subscription scheme, for example, `/e-Auction/*/ScienceFiction`. Figure 2.7 illustrates the subject-based subscription scheme for an auction site: The subscriber S_2 subscribes to the channel `/e-Auction/Books` and receives only the notification e_2 published on that channel. The subscriber S_1 is subscribed to the channel `/e-Auction` and thus receives all notifications published on that channel including the notifications published on channels `/e-Auction/Books` and `/e-Auction/DVDs`. Compared to channels, subjects offer better expressiveness, although expressiveness is still limited because subjects offer a static view of notification categories. Secondly, a large number of subject subcategories may lead to the explosion of a subject hierarchy tree which is a serious implementation problem, especially if the implementation is based on IP multicast: Each sub-subject must be implemented as a special multicast group and the number of multicast addresses is limited.

Content-based subscription. The alternative approach to the static channel-based scheme is the content-based subscription which offers a more sophisticated and flexible subscription scheme

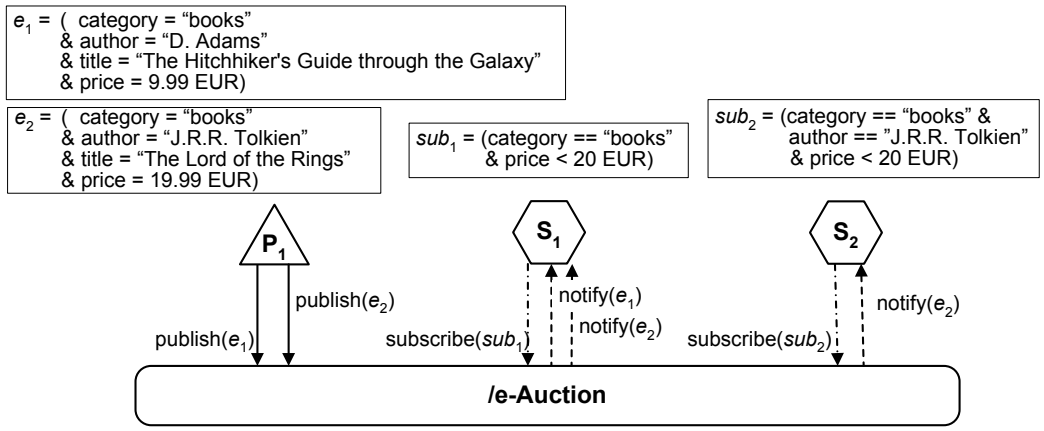

Figure 2.8: Content-based subscription scheme

with increased expressiveness. It enables subscribers to define properties of notifications they are interested in: Each subscription is a predicate which can test notification properties, i.e., attribute-value pairs. A subscription predicate consists of a sequence of patterns that combine notification attributes with a subscription constraint that defines the attribute value. The content-based subscriptions enable subscribers to describe the properties of notifications along multiple dimensions ensuring flexible and subscriber-centric notification filtering. The filtering minimizes the number of uninteresting notifications delivered to subscribers.

Figure 2.8 illustrates an example content-based subscription to the subject `/e-Auction`. The notification space is modeled as a single subject and subscription predicates are used to specify subscriptions. For example, the subscriber S_1 is interested in books below 20 EUR. It specifies the subscription using the predicate `(category=="books" & price<20 EUR)`. The subscriber S_2 is also interested in books below 20 EUR, but only in those written by J.R.R. Tolkien. It specifies the predicate as `(category=="books" & author=="J.R.R. Tolkien" & price<20 EUR)`. The content-based scheme offers increased expressiveness when compared to the subject-based scheme: To offer the same type of subscription expressiveness in the subject-based scheme, new subjects `/e-Auction/Books/LessThan20_EUR` and `/e-Auction/Books/Tolkien/LessThan20_EUR` would be created making the subject tree quite complex. Moreover, it is not possible to predict the preferences of all subscribers and to create static subjects that match all their needs. The dynamic nature of the content-based subscription scheme is obviously superior with respect to expressiveness. However, it complicates the implementation of the publish/subscribe system which has to deal with subscription predicates [23]: The design of an efficient and scalable solution to the problem of matching notifications to subscriptions is still an open problem under active research [20, 78].

Type-based subscription. The type-based subscription scheme is a static classification scheme which resembles the subject-based approach. It uses types from object-oriented languages for distin-

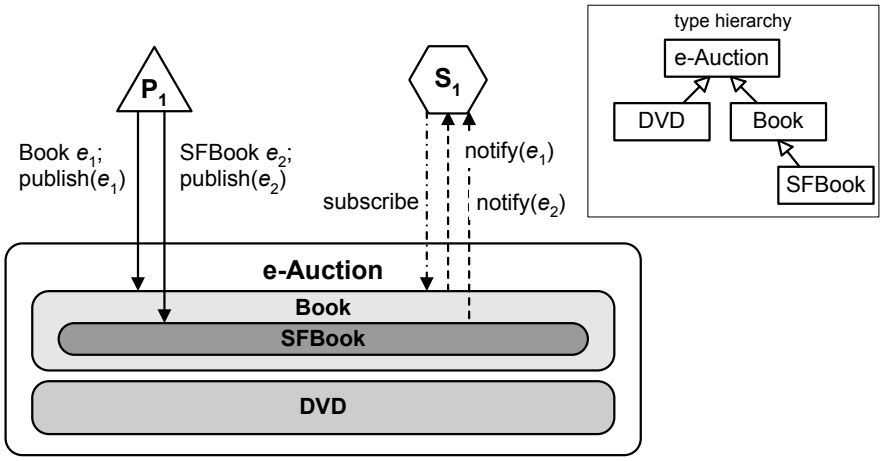

Figure 2.9: Type-based subscription scheme

guishing notification categories [45]. In type-based systems notifications are modeled as objects, and each notification object has a type. The notion of notification subject is matched to the notion of notification type, and the notification hierarchy found in subject-based subscriptions is mapped to the type inheritance tree found in object-oriented languages. Subscribing to a type implies that a subscriber receives all notifications of the class implementing the type and also all notifications of its inheriting subclasses: In other words, a number of classes may conform to a single type. For example in Figure 2.9 the subscriber S_1 subscribes to the type `Book` and receives the notification e_1, an instance of the class `Book`, and also the notification e_2, an instance of the class `SFBook` which inherits the class `Book`.

The main advantage of the type-based subscription over the subject-based scheme is that subject hierarchy is naturally mapped to the inheritance tree found in object-oriented languages. The process of matching a notification object to a type is performed by querying weather the notification object is an instance of a defined type. When compared to the content-based scheme, the type-based approach ensures the encapsulation of data within notification objects which is violated in content-based systems.

It is possible to combine either the subject-based or the type-based subscription scheme with the content-based scheme: Static classification of the subject-based or the type-based scheme is extended by the dynamic nature of the content-based scheme. Clearly, the matching problem – determining the subset of all subscriptions with predicates that match each published notification [5] – is the major bottleneck in the content-based systems used in realistic applications with a large number of published notifications and subscriptions. A naive matching algorithm would match each published notification against every existing subscription. This algorithm runs in time linear to the number of subscriptions and becomes inefficient in environments with a large number of publishers and subscribers [5]. The main rationale for improving this naive approach is that it is highly probable for several subscription

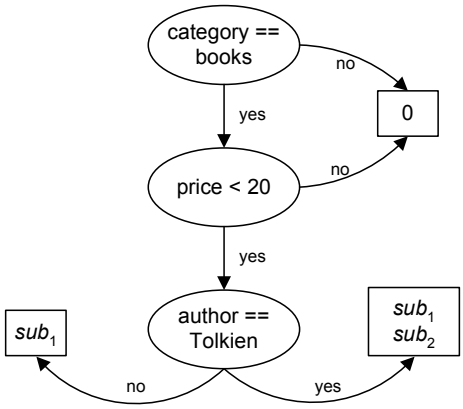

Figure 2.10: Decision tree for a content-based subscription

predicates in a publish/subscribe system to share some common sub-expressions. It is, therefore, reasonable to exploit the commonality and check each expression only once. Subscriptions are combined into decision trees that enable simultaneous matching of notification properties to subscriptions with common constraints on notification attributes. Decision trees are updated with each new subscription which is adequate for the environments in which the frequency of published notifications is very high when compared to the frequency of subscription changes.

Two subscriptions from the example in Figure 2.8 can be combined into a decision tree illustrated in Figure 2.10. The decision tree consists of three nodes, each representing an expression that tests a notification attribute. The rectangles represent end states with a list of subscriptions that match a published notification. When a notification is published, its attribute-value pairs are matched against node expressions, until reaching the end state with a list of subscriptions that identify the subscribers to which the notification must be delivered.

Construction of a decision tree is not a trivial task, especially when the number of subscriptions is high. Reference [20] lists the requirements for building an efficient filtering engine, and proposes an approach based on binary decision diagrams. A pragmatic solution for implementing a content-based publish/subscribe service using structural reflection is presented in [43].

2.3.3 Characteristics of Publish/Subscribe Systems

In this section we summarize the important characteristics of publish/subscribe systems.

Loose coupling and system extendibility. One of the major characteristic of the publish/subscribe interaction model is its inherent loose coupling. The communicating parties are temporally and referentially decoupled since they interact through an intermediary. The communicating parties can frequently be deactivated and reactivated, without affecting the functionality of the system as a whole. Furthermore, the integration of additional components does not affect the functionality of the existing components in the system which ensures system extendability.

Filtering. Filtering of published notifications ensures that a subscriber receives only the notifications that satisfy certain criteria defined in its subscription. This approach minimizes network traffic and the processing done by a subscriber. It is attractive to perform the function of notification filtering close to notification sources especially when network resources are scarce. This process is usually performed by an intermediary.

Notification persistency. The employment of an intermediary which is usually operating permanently on a stationary host can ensure notification persistency. The intermediary can store the received notifications prior to their delivery to subscribers, and discard them after receiving an acknowledgment from all notification receivers. If a subscriber is inactive, the notification can be stored and delivered when the subscriber is activated. To ensure notification publishing, a publisher can store a published notification locally until the intermediary acknowledges notification receipt. Hence, the model is adequate for designing a system with delivery guarantees: *At-least-once* and *exactly-once* delivery guarantees can be implemented by incorporating acknowledgments into the basic communication protocol. *Totally-ordered* notification delivery may be ensured using notification sequence numbers per each publisher.

Scalability. The architecture of the notification service can be either *centralized* or *distributed*, depending on the scope of the system, the number of publishers and subscribers, and the number of published notifications. Centralized implementation might become a bottleneck for wide-area systems. In such cases the distributed service architecture is applied.

The distributed publish/subscribe service usually comprises a number of *brokers* that form an overlay network. Each broker is in charge of the publishers and subscribers that are connected to it. It is also responsible to submit the published notifications to other brokers having subscribers interested in them, to accept the notifications coming from other brokers, and to deliver them to their subscribers with the matching subscription. Brokers need to exchange control messages so as to have a consistent view of advertisements and subscriptions in the system. They maintain a routing table with the routing information for forwarding published notifications to interested subscribers. For example, if a notification is published and its subscribers are connected only to the local broker, this notification will not be delivered to other brokers in the system.

The distributed publish/subscribe service architecture solves the scalability problem and offers the means to design a fault tolerant system. However, the design of a distributed architecture is more complex than of a centralized solution, and it introduces significant communication load due to the exchange of control messages between system brokers. Hence, it is necessary to adjust the employed publish/subscribe system design to traffic requirements of the actual application setting.

2.4 Mobility

Mobility is the ability of moving readily in space. With respect to communication and data services, it is the ability of an end user to move through different geographical locations and to use different terminals in different networks, subnetworks and network domains while applying the service.

Firstly, the applied network, or a number of networks, need to support the mobility of terminals while providing access to the Internet. The wireless part of the network assumes cellular topology. A mobile terminal moves within radio cells and attains its network connection through a cell base station using radio signals. The terminal can move between different cells and maintain a continuous network connection using the mechanism know, as a handover, which transfers the connection from the old base station to a new base station [47]. There are a number of solutions that provide wireless IP access. Wireless LAN (WLAN) technology is based on the IEEE 802.11 LAN standard that defines new physical and data link layers to provide wireless connectivity within IP-based LANs [63]. Mobile networks from telecommunications domain, such as GPRS [103] and UMTS [60], rely on packet switching technologies to support data services for mobile terminals. Wireless networks unlike wired networks exhibit considerable constraints, such as lower bandwidth, higher latencies, and intermittent connectivity.

Secondly, the used terminals need to be mobility-enabled, i.e. small, portable and capable of preserving the connection to the network at different geographical locations. Mobile devices are resource-scarce when compared to their stationary counterparts [104]. Computational resources, e.g., processor speed, memory size, disk capacity, and display size are inferior and costly because of the size and weight limitations. Next, mobile terminals run on batteries with a limited power supply. Furthermore, the wireless network bandwidth is always lower and more variable than in wired environments.

Thirdly, the service placed in the network must be mobility-aware and flexible to support a constantly changing number of terminals. The service should adjust to specific needs of applications running on mobile terminals. Such applications should be context-aware and adaptive [73]. The application must react to the changes in the environment and on the device, e.g., bandwidth variations, frequent disconnections, and the status of resources on the terminal, and adjust its behavior based on this knowledge.

2.4.1 Mobility-Aware Content Dissemination

We assume that the content dissemination service supports mobility of end users, both content publishers and subscribers, in diverse networks. The service is placed in an IP-based network and executes in an extremely dynamic context. Publishers and subscribers change network access points, network domains, and terminals in different networks. The characteristics of network domains and network access points differ significantly. The availability of services in different networks varies, and the availability of terminal resources such as memory, bandwidth, or battery power is constantly changing. Therefore, the service must be capable of adapting to the changes in both the network and

terminal context, and has to deal with different types of mobility.

Figure 2.11 depicts an environment for the deployment of a content dissemination service. Publishers and subscribers may roam in a network domain which uses WLAN as a radio access network (RAN), or in a GPRS/UMTS wireless domain. The WLAN domain is IP-based and adequate for the packet-based transport of the published content with theoretical maximum bit rate of 54 Mbps for IEEE 802.11a WLAN and 11 Mbps for IEEE 802.11b WLAN. The GPRS network provides packet data transport at the rates from 9.6 to 115 kbps (theoretical maximum 171 kbps), while UMTS offers theoretical maximum bit rate of 2 Mbps, which in reality falls down to 384 kbps or 144 kbps for mobile terminals moving at high speed. WLAN enables wireless broadband access over smaller geographical areas, the so-called “hot spots”, while GPRS and UMTS networks offer mobility over wide coverage areas.

In GPRS/UMTS the base stations are grouped and controlled by a base station controller (BSC) – radio network controller (RNC) in UMTS – that comprises a RAN. GPRS enables mobile terminals to communicate with terminals located in the external IP networks. GGSN is a gateway used as an interface to an external packet data network such as the Internet. The tendency is to introduce IP-based data transport close to a mobile terminal, and to build all-IP networks. For example, GPRS uses IP in its backbone network that connects serving GPRS support nodes (SGSN) with a gateway GPRS support node (GGSN). However, IP is not used in its native mode for routing data packets between SGSN and GGSN: A special protocol, GPRS Tunneling Protocol (GTP), built on top of UDP/IP tunnels the packets between SGSN and GGSN nodes. The major difference between GPRS and UMTS architectures is in their RANs: Time division multiple access (TDMA) is used for air interface transmission in GPRS networks, while wideband code-division multiple access (WCDMA) is applied in UMTS networks. In its packet switched core network, UMTS release ’99 architecture uses the elements that have evolved from GPRS networks. The next UMTS releases, release 4 and 5, are gradually evolving UMTS networks into a converged packet based network [71]. The main reasons for the all-IP approach are lower infrastructure and maintenance costs of a single converged network and easier deployment of novel services in IP-based networks. The aim is to provide real-time multimedia services that impose strict QoS requirements on network performance with respect to packet loss and delay. This is still a challenge in packet-based networks where network resources are not reserved ahead of a multimedia session.

Service environment is highly dynamic due to mobility of terminals and users, intermittent network connectivity, variable bandwidth, and the inherent probabilistic nature of the content publishing service itself. It is dynamic from the service point of view because publishers deliver the content for publication at random, and subscribers change subscriptions. Furthermore, they move across the network, and change access points and network domains. Publishers and subscribers are also affected by the dynamic nature of the environment. As they move between networks and domains, the available services change, and they need to be aware of service availability in the particular visiting domain. Service architecture must deal with the constant changes in the number, location, and properties of the participating publishers and subscribers and operate in a heterogeneous environment with a diversity

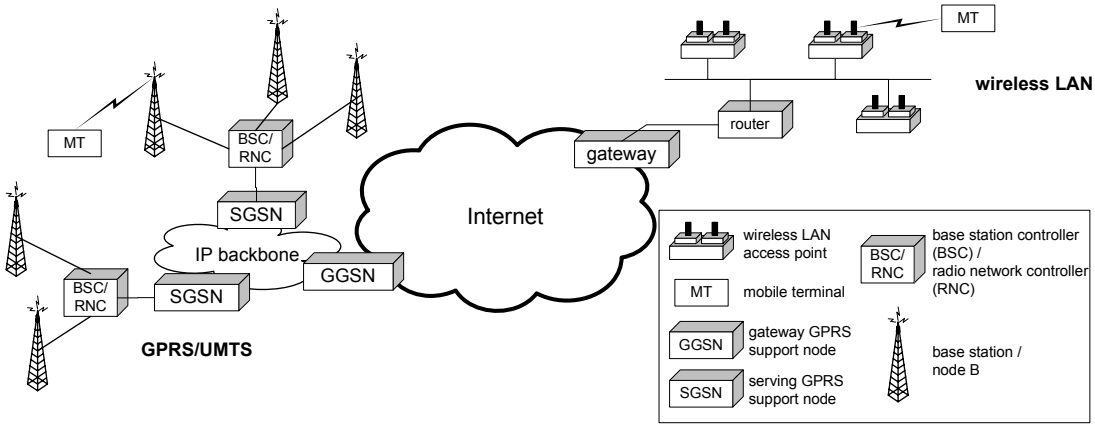

Figure 2.11: An environment for service deployment

of wireless access technologies and terminals.

Distribution transparent interaction models, such as RPC/RMI, are inadequate for highly dynamic and heterogeneous mobile environments. Temporal coupling and transient communication style cannot be applied in environments where communicating parties are often disconnected and unreachable. Furthermore, transport protocols, such as TCP and UDP, expose weaknesses in mobile environments, because the characteristics of wireless links can significantly affect their performance. TCP assumes that packets are dropped due to network congestion, and forces senders to lower the transmission speed which deteriorates TCP performance. UDP packets are often dropped in wireless environments and the application or middleware using UDP must deal with the unreliability of data transport.

The traditional requirement for distribution transparency is found unsuitable for distributed interactions that are unreliable and introduce significant latency frequently found in mobile settings [73]. *Loose coupling of communication parties* and *context-awareness* are promoted as key characteristics of mobile middleware. Temporally and referentially uncoupled systems offering persistent communication can deal with intermittent connections, terminal unavailability, and changes of terminal network access points. Context-aware middleware makes the information about the execution context available to higher layers, and enables applications and end users to take actions when the context changes.

Communication patterns put/get and subscribe/publish/notify offer persistent and temporally decoupled communication adequate for mobile environments. The put/get interaction style is designed for point-to-point communication, while publish/subscribe is inherently multi-point, and suitable for the implementation of content dissemination services. Asynchronous, anonymous, time-independent, and persistent characteristics of the publish/subscribe interaction model make it adequate for highly dynamic mobile environments [30]. Publishers and subscribers are fully decoupled: They interact with the service without the knowledge about other information sources and sinks. The publish/subscribe model supports system extendability: The addition of a new publisher or subscriber does not affect system functionality which is desirable when dealing with frequent disconnections.

Publish/subscribe systems also satisfy the requirement for context-awareness: The event-based nature of the publish/subscribe interaction model offers means for designing responsive applications aware of publisher's and subscriber's states, and adaptable to the changes in network context.

2.4.2 Mobility Management

Content dissemination services require the transport of non-realtime data and do not impose strict real-time QoS requirements on the network infrastructure. The service can benefit from higher data bit rates, for example, those offered by WLAN or UMTS, however, the existing GPRS networks can offer sufficient bandwidth for the transport of published notifications. Nevertheless, a higher bit rate alone is not sufficient for the deployment of mobile content dissemination services that support personal mobility of end users. The network needs to provide *terminal and personal mobility management* to ensure the mobility of terminals and people in different networks, and to deliver the content to mobile terminals that are at a particular moment used by service subscribers. Various mobility management procedures are used in telecommunication networks and in the Internet. Mobile telecommunication networks offer efficient solutions for seamless terminal mobility and handover [6]. The Internet, on the other hand, was not originally designed to support mobility of terminals and users. Mobile IP [88] is currently a prevalent solution for terminal mobility in the Internet domain. It is necessary to combine the mobility management solutions from both domains, and to design a solution for personal mobility management since it is not available in the existing networks.

Mobility management is a network service that enables a mobile network to locate users applying mobile terminals across different network domains, and to maintain an active session when mobile terminals change service domains. Mobility management is essential for services that initiate a communication session to a user, such as a voice call, a multimedia session, or a push-based content delivery. A current network access point of a mobile terminal, such as an IP address, or the identifier or a wireless cell, is needed to deliver data packets or to set up a multimedia session with a user. The network needs to store the up-to-date information that can enable quick and efficient localization of users and terminals. Mobility management comprises two distinct services: *location management* and *handover* [7]. Location management stores and updates the information about current network access point of a mobile terminal or a user. Handover enables a mobile user and terminal to roam across different network cells and domains, and maintain an active session regardless of terminal or user movements.

With respect to the type of mobile entity, mobility management schemes are applied for different types of mobility, such as *terminal*, *personal*, or *session* mobility [106]. Terminal mobility enables mobility of a single terminal within a mobile network. Personal mobility deals with mobility of end users who apply different terminals in various networks. Session mobility enables seamless transfer of an ongoing communication session from one terminal to another without the loss of information. Terminal mobility management is an essential service in today's mobile and wireless networks. Problems and solutions related to terminal mobility are widely explored in the literature [6, 7, 88]. The

need for personal mobility management that regards a user, rather than a terminal, as an end communication point, extends the concept of terminal mobility from a single terminal and network, to various terminals in different networks [101]. Currently, there are no standard solutions for personal mobility that are widely applied in today's networks. Session mobility has been addressed recently.

Terminal mobility. Terminal mobility deals with the mobility of a single terminal in a mobility-enabled network. A terminal can seamlessly move and attain the connection to the core network, despite changed location and the applied network access point. We distinguish between nomadic and truly-mobile terminal mobility.

Nomadic terminal mobility enables network connectivity from arbitrary and changing locations, but there is no network connectivity during the move procedure. A nomadic user may disconnect from a network access point, move to another network domain, and reconnect through another access point. Terminals may change access points only between successive data sessions. The handover procedure for nomadic mobility does not impose demanding requirements on the network infrastructure. It is even possible to use a wired network connection for this mobility scenario.

Mobile IP [88] is nowadays a prevailing solution for nomadic mobility in IP-based networks. It is a network layer protocol that supports *macro-mobility* across the Internet. The essential problem when dealing with mobility in IP-based networks is that IP-based protocols are designed for stationary terminals. An IP address is used to identify a terminal and its location in the Internet. When a mobile terminal connects to another network domain, it needs to use a new address from the new domain to be addressable in the Internet. Mobile IP enables a mobile terminal to be uniquely addressable through a permanent home address, regardless of its position in the network. When a terminal migrates to a new network domain, it acquires a temporary care-of-address from a new domain and registers it with its home agent in the home network. The home agent intercepts the packets sent to a mobile terminal and tunnels them to the new care-of-address. Mobile IP uses the *home-based approach* to location management [91]: The home agent maintains a mapping between a mobile terminal and its current IP address. The problem with this approach is that the assignment of a home agent to a mobile terminal is permanent, and it introduces performance overhead due to triangular routing. Moreover, packet tunneling between a home agent and a mobile node adds overhead which is significant for bandwidth-constrained wireless links. This overhead and the registration of each new care-of-address with the home agent introduces high latency during the handover, and makes mobile IP inadequate for real-time handover of quickly moving terminals. Mobile IPv6 offers a solution to triangular routing using dynamic binding between a terminal's permanent and temporary address at the correspondent host, which decreases the bandwidth consumption and communication latency.

Truly-mobile terminal mobility assures network connectivity during terminal movements between wireless cells and offers greater movement flexibility than nomadic mobility. The essential infrastructure for truly-mobile terminal mobility is the availability of a mobile network that offers seamless handover support. The network enables a mobile terminal to change a network access point during a session in real time and to perform the handover procedure that is not observable by an end user.

Mobility management mechanisms for mobile networks have been designed to support seamless

terminal mobility [70]. Their key advantage over the approach found in IP networks is in the concept that separates the identifier of a mobile terminal from its location in the network, and the applied network access point. A network maps the unique terminal identifier to its current location in the network using the *two-tiered scheme* for location management [91], a Home Location Register (HLR) and a Visitor Location Register (VLR). HLR maintains the current location of a mobile terminal together with a user profile. VLR is associated with a predefined area and maintains the information about the terminals currently visiting its area. In the mobile-controlled handover, a mobile terminal monitors the signals of the surrounding base stations, and initiates the handover when a signal from the currently serving base station deteriorates. It can perform location update either periodically, or each time it enters a new predefined location area comprising a set of base stations. The change of location area is reported to VLR, and if the mobile terminal enters the area associated to a new VLR, the change is reported to HLR. The two-tiered scheme found in mobile networks resembles the home-based approach of mobile IP because of the home entity that stores the information about the whereabouts of mobile terminals. The main difference between the two approaches is that the location query in mobile IP is always directed through a home agent, while in HLR/VLR approach the search for a mobile terminal starts with a local VLR, and if the local VLR has no entry about the mobile terminal, the query is directed to HLR.

To use data services in GPRS/UMTS networks, a mobile terminal needs to register with the network using the *attach procedure* [103]. The terminal sends an attach request to SGSN responsible for current terminal location area, SGSN checks the terminal's subscription data stored by HLR, and grants access to network services. Furthermore, the *PDP context activation procedure* enables a mobile terminal to acquire an IP address that is used for routing data packets to the terminal from the external packet data networks. GGSN maintains the mapping between the terminal's address and its serving SGSN, and tunnels the packets received from the external network to SGSN that delivers them to the mobile terminal. The handover procedure is called *packet rerouting*: It occurs when the mobile terminal changes its serving SGSN and causes the rerouting of packets from GGSN to the new SGSN. The correspondent host from an external network is not aware of the handover since the terminal maintains its external identifier. Mobility management in GPRS networks is also based on the two-tier scheme: GGSN is a gateway that maintains the mapping between the external terminal identifier and its current location in a GPRS network.

Micro-mobility solutions for IP-based networks have been designed to solve the problem of significant handover delay of mobile IP [99]. The main idea of micro-mobility is to use mobile IP for inter-domain mobility, but to apply a different procedure when a mobile terminal moves within a single administrative domain. The change of the location within the domain causes the change of the terminal care-of-address, but this information is maintained within the domain, and is not propagated to the home agent that remains unaware of terminal movements within a domain. This approach follows the ideas used in GPRS networks since a domain root router supporting micro-mobility is the only visible element from external networks that keeps the identifier of a mobile terminal fixed while it moves within its domain, and maintains a current mapping to the location of the mobile terminal

with the domain.

Personal mobility. Personal mobility regards end users as end communication points. A user can employ various terminal devices in different networks and stay uniquely addressable within the scope of a service. Personal mobility reflects the need to locate people and establish a session to a user through a terminal. For example, a unique user identifier is mapped to one of the temporary terminal addresses, e.g., email address, mobile phone number, or service-specific address, such as ICQ number. The mapping enables a service to adjust the communication means to the current application, terminal, and user preferences that define conditions under which certain communication means are adequate. Personal mobility management can be regarded as a layer on top of the existing terminal mobility schemes. It requires support for terminal mobility within the existing networks, and extends it by a location management procedure that maintains a dynamic binding between a uniquely addressable user and its current network attachment point.

Mobile People Architecture (MPA) [72] is a solution for personal mobility that aims at enabling person-to-person communication while preserving mobile person's privacy and communication preferences. It proposes a *person layer* on top of the application layer found in traditional networks by adding a new entity to the network, a *personal proxy*. Personal proxy tracks current application and the means for user reachability, intercepts incoming communication sessions and data, and directs and adjusts them to currently used application. For example, a voice message can be converted into an e-mail message if a user is currently reading mail and has decided not to accept phone calls.

Personal mobility is closely related to the *presence service* specified in the context of instant messaging [34]. A presence service accepts, stores, and distributes presence information about mobile users - presence entities. The presence information describes the current status of a presence entity regarding its communication capabilities, for example, a user can be "online" or "offline". Presence information is given in the form of XML documents that contain the information about communication means, user preferences, and current geographical location. Presence services are event-based: Presence entities provide their presence information to the presence service. Presence service accepts, stores, and distributes the presence information to everyone subscribed to get the notifications about presence changes.

Session mobility. Session mobility enables users to maintain an active session while changing terminals. A session is moved from one terminal in one network to another terminal in another network while preserving an ongoing session. This concept requires migrating current service execution related information, including service context and service data, from one terminal to another.

Session Initiation Protocol (SIP) [105] is a signaling protocol for multimedia services that enables setup and management of multimedia sessions. The network must primarily locate an end user to setup a session. SIP is an application layer approach that offers the means for terminal, personal, and session mobility. SIP binds a user-specific identifier to a temporary IP address of a terminal that is currently applied by a user and inherently provides means for personal mobility. Users can maintain the same identifier as they change network attachment points or use different devices. Users register their temporary location in the network to a home SIP redirect server or leave a pointer to other servers

Table 2.3: Requirements for mobility management

	nomadic	**mobile**
single terminal	terminal mobility ○ *mobile IP*	terminal mobility ○ *micro-mobility for IP networks* ○ *mobility management in GPRS/UMTS*
various terminals	personal mobility ○ *MPA*	terminal & personal mobility ○ *SIP*

where the information about the user location may be found. The distributed network of SIP registrars and redirect servers implements a home-based mobility management procedure. A home registrar receives a request for session setup, and initiates the process for locating the user following pointers at a number of redirect servers. On top of personal mobility, SIP offers the support for session mobility by enabling a redirection of an active multimedia session.

Terminal and personal mobility management are vital for the operation of content dissemination services. Session mobility is not an issue because the service disseminates non-real time content. A number of scenarios for service usage are possible. A user may be either nomadic or truly mobile, and apply a single or various terminals for content receipt. Different mobility management solutions are needed for the four usage scenarios as listed in Table 2.3.

For a single device moving across domains we can assume the usage of mobility management protocols for terminal mobility. Mobile IP is adequate for nomadic users, while micro-mobility solutions for IP-based networks and terminal mobility management procedures found in GPRS/UMTS can be used for truly mobile terminals. In usage scenarios where users apply a number of terminals for content receipt, the support for personal mobility is required. If a user is nomadic a personal mobility service is required to authenticate the user and track the current identifier of the applied terminal, similar to the MPA approach. In a mobile scenario with multiple terminals both terminal and personal mobility management are needed. SIP is currently the only service providing such support. However, SIP is primarily designed to support multimedia services and its scalability and performance need further investigation if applied as a terminal and personal mobility management solution. The argument in favor of applying SIP for personal mobility is that SIP has been selected as the main signaling protocol in future all-IP mobile networks [2]. It might become widely available and extended to support non-realtime traffic.

Chapter 3

Related Work

Scalable and efficient dissemination of content to users residing in wide area networks has been an area of active research for many years. Different approaches have been designed ranging from network-level to application-level solutions. IP multicast is a network-level solution for the efficient group communication. Services like electronic mail and Usenet news are the established applications used for communication and exchange of content in everyday life. Recently, push systems and publish/subscribe systems have been introduced to provide active content dissemination with increased content customization through user subscriptions. The listed systems are mainly designed for stationary environments. However, the development of higher bandwidth mobile networks promotes the deployment of content dissemination services in mobile environments. The widespread use of SMS, and the increased interest in MMS clearly indicate the demand for information delivery that enables user mobility.

This section gives an overview of the concepts and systems related to content dissemination services. In section 3.1 we analyze the existing publish/subscribe systems, and explore the publish/subscribe concepts related to mobility in Section 3.2. Section 3.3 considers the related approaches. First, we briefly discuss electronic mail and Usenet news, the established applications from the Internet domain. Next, we analyze SMS and MMS, services for information dissemination in mobile networks. Finally, we list the basic properties of application-level multicast systems and push systems.

3.1 Representative Publish/Subscribe Systems

A number of solutions and systems based on the publish/subscribe interaction model are in use today. CORBA event service [82] and CORBA notification service [83], as defined by OMG, are implemented in a number of CORBA systems. Sun has developed the Java Message Service [108] specification that incorporates the publish/subscribe principles. TIBCO's TIB/Rendezvous [116] is an example of a commercial system widely used in business applications. There are also a number of research projects, e.g., JEDI [19, 32, 33], Siena [23, 24], DACs [44, 45, 43], Hermes [89, 90], and

REBECA [49, 78], that are designed and implemented using the publish/subscribe concepts.

The research in the area of publish/subscribe systems has thus far concentrated on the design of an adequate subscription language with an efficient filtering engine [5, 20], topology and routing for distributed architecture [24], or the quality of service mechanisms for reliable and ordered content delivery [126]. Reference [42] offers a systematic analysis of the publish/subscribe characteristics. Reference [33] classifies and compares the existing publish/subscribe systems. We present a short overview of the established publish/subscribe systems. A detailed discussion and comparison is given in [92].

3.1.1 CORBA Event and Notification Service

Common Object Request Broker Architecture (CORBA) incorporates the support for the event-based publish/subscribe interaction between CORBA objects. CORBA event service [82] defines a set of publish/subscribe service interfaces and describes the underlying infrastructure: Objects can publish notifications or be interrupted upon the occurrence of a particular notification. The notification service specification [83] extends the event service by advanced facilities, such as quality of service and notification filtering.

Event service. CORBA event model [82] follows the channel-based publish/subscribe communication pattern: Objects communicate via *channels* that allow multiple suppliers to communicate with multiple consumers in an asynchronous way. *Suppliers* connect to a channel to publish notifications, denoted events in the context of CORBA event service, and *consumers* connect to the channel to receive the notifications.

In a simple scenario consumers and suppliers can interact directly, without a channel, by invoking each other's interface methods. A mediated scenario involves a channel that acts as both a supplier and a consumer of events. Event channels enable anonymous many-to-many communication between suppliers and consumers. However, event channels offer no means for event filtering: All channel consumers receive all events published on the channel.

CORBA event service supports both *push* and *pull* approaches to communication initiation: The push model allows suppliers to initiate the distribution of notifications to consumers. The pull model allows consumers to request notifications from suppliers. Push suppliers actively send notifications to the event channel, while pull suppliers wait for requests coming from the channel. Push consumers passively wait for events that are eventually sent through the channel, while pull consumers regularly check if new events are available on the channel.

CORBA notifications are associated with a single data item, such as an object reference, or an application-specific value, but they are not objects, since CORBA does not support passing objects by values. Two orthogonal approaches to notification communication are defined: generic, and typed. Generic notifications carry messages with an undefined structure. In the typed case, notification data is passed by means of typed parameters that are defined by the Interface Definition Language (IDL). Accordingly, event channels can also be generic or typed: Generic event channels only support generic

communication, while typed event channels support both typed and generic communication. One of the major constraints of the approach is that suppliers and consumers need to agree beforehand on the structure of notifications so as to be able to process them in a meaningful way.

The event service defines simple means for event propagation and has a number of drawbacks. It is not adequate for mobile scenarios because consumers must be connected to the channel at the time of event publication because CORBA event service does not support event persistence. The second drawback is that event channels offer no means for event filtering: If various event types are needed, it is necessary to use separate channels for each event type. Finally, the specification does not dictate the reliability requirements for the communication service and offers no guarantees concerning the delivery of events. It can have either "at-most-once" or "exactly once" semantics, depending on the particular service implementation.

Notification service. CORBA notification service [83] deals with the mentioned drawbacks of the event service and extends it with new capabilities, such as filtering and configurability, according to various requirements for quality of service (QoS). The notification service preserves the semantics of the event service and ensures interoperability between the basic event service clients and notification service clients. One of the extensions offered by the notification service are content-based subscriptions and event filtering using filter objects: Filter objects define a set of constraints that affect the forwarding of an event. For example, notification service consumer can subscribe to events of interest by associating a filter object to the proxy through which it connects to an event channel. When an event that matches the filter object is published, the proxy will forward it to the consumer.

The notification service introduces a new type of events, *structured events* with a well-known data structure into which a wide variety of event types can be mapped. Structured events consist of a header and a body: A *header* is further decomposed into a fixed and a variable part. The fixed event header consists of a `domain_name` which identifies a particular domain (e.g. telecommunications, finance), a `type_name` which categorizes an event, and an `event_name` which can uniquely specify an event. A variable header part is composed of a list of optional name-value pairs. *Event body* carries the content of an event. The filterable portion of the body contains the most interesting event fields (name-value pairs) used when matching the event with a filter object. The remainder of the body is of type `any` and can be used to transmit large data items.

Another enhancement introduced by the notification service are standard interfaces for controlling QoS characteristics for event delivery. The notification service enables each channel, each connection, and each message to be configured so as to support the desired quality of service with respect to delivery guarantees, event persistence, and event prioritization. OMG defines a set of QoS properties, their permitted types, and the range of values. This is an open list of parameters, and service implementers can add their own properties. OMG has defined the following properties:

- Reliability is related to the event delivery policy, such as best effort, or persistent delivery.
- Priority; by default, the notification channel will attempt to deliver messages according to their priority level.

- Expiry times indicate the time interval within which an event is valid.
- Earliest delivery time specifies the time after which an event can be delivered.
- Maximum events per consumer defines the maximum number of events a channel can queue on behalf of a consumer. This property prevents malicious users from overloading a channel.

The list of supported properties provides flexible QoS configuration of a notification channel. However, meaningless properties are not prevented which creates a serious vulnerability that could be exploited by malicious consumers or suppliers. End-to-end delivery policy can only be guaranteed with the cooperation of all parties, i.e., consumers, suppliers, and the notification channel.

The OMG event and notification service specifications offer no guidelines regarding the architecture and routing strategy for distributed event systems.

3.1.2 Java Message Service

Java Message Service (JMS) [108] is a message-oriented specification for the Java programming language that defines a set of interfaces and their semantics, thus enabling JMS compliant clients to access the services offered by a JMS messaging server. JMS target application area is enterprise messaging for asynchronous Business-to-Business communication over the Internet. JMS provides two types of messaging models, point-to-point messaging and publish/subscribe [76]. The point-to-point messaging model relies on the classical message-queuing communication pattern. The publish/subscribe model incorporates two types of JMS clients, publishers and providers, that communicate by exchanging messages through an intermediary server, called JMS provider. A *JMS provider* is a messaging server that implements JMS interfaces and provides administrative and control features. *JMS clients* are programs or components written in the Java programming language that produce and consume messages. Notifications are referred to as *messages* in JMS: Messages are Java objects that communicate information between JMS clients.

The concept of notifications is circumvented in JMS and used implicitly in messages. Message properties and its payload represent a notification, i.e., information submitted to message subscribers. Messages consist of message headers, message properties, and message data called the payload, or message body. Message headers carry message routing information, and control information about the message, such as message id, timestamp, priority, and delivery mode. Message properties consist of attribute-value pairs and can be defined by the application using JMS, or by the messaging server. Message payload contains the information that message publishers communicate to message receivers.

Publishers publish messages to a *JMS topic*, which is one of JMS destinations. Topics are created by an administrator using the administrative tools offered by the applied JMS provider. It is assumed that publishers will publish messages on the established topics. This approach is static and is augmented by *temporary topics*: Publishers can dynamically create new temporary topics. Subscribers subscribe to a particular topic by registering their message listeners with the topic, as depicted in Figure 3.1. Whenever a message is published on a topic, the listener's method is invoked, signaling the

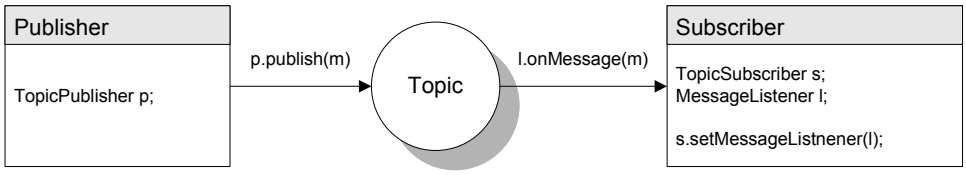

Figure 3.1: Publish/subscribe interaction in JMS

receipt of a new message for the subscriber.

JMS offers delivery guarantees using the concepts of *durable subscriptions* and *persistent messages*. Subscribers can define durable subscriptions to a topic. While a durable subscriber is disconnected from a JMS server, the server stores the published messages matching its subscription. When the subscriber reconnects, the server sends all stored and unexpired messages to the subscriber in the *store-and-forward* delivery style. Publishers can define either persistent or non-persistent delivery mode for their messages. In case of the non-persistent mode, the server offers at-most-once message delivery. Persistent messages are first stored by the server, and then delivered to subscribers. Subscribers need to confirm the receipt of a persistent message. If the acknowledgment is missing, the server resends the message assuring at-least-once message delivery.

JMS defines *message filtering* on the subscriber side using message selectors. Message selectors are expressed as Java strings that define conditions on message properties and headers. Message selectors need to comply with the defined subscription grammar which supports the conditions as complex boolean expressions with equality, comparison, or range operators. The JMS specification is a pure API specification. It does not define the rules for building the architecture of JMS server with respect to distribution.

3.1.3 TIB/Rendezvous

TIB/Rendezvous is a commercial messaging system for application-to-application integration that is based on the publish/subscribe communication pattern [116]. It is originally described in terms of an *information bus* [85] that offers application independent communication through self-describing messages. TIB/Rendezvous is a widely used messaging system applied for integrating diversity of applications, e.g., various financial and banking applications.

TIB/Rendezvous adopts the subject-based approach to subscriptions. A subject name is a sequence of strings separated by dots (e.g., `news.comp.theory.books`) that arrange subjects in a subject tree. A subscriber can subscribe to a single subject, or use wildcard characters, such as "*" (substitutes a single string) or ">" (substitutes a number of strings) to specify a range of subjects. Messages communicated between publishers and subscribers are composed of a set of typed data fields. A field is a record with the following attributes: `name` specifies the name of the field, `ID` defines a message-unique field identifier, `size` gives the total size of the field, `count` contains the number of elements if the field consists of an array, `type` indicates the type of field data, and, finally,

`data` contains the actual data stored in the field [115].

TIB/Rendezvous employs a distributed architecture to offer reliable and scalable distribution of notifications using different transport mechanisms, such as link-level network multicasting, IP multicasting and point-to-point communication. Each host running a client, either a publisher or a subscriber, must run a special *rendezvous daemon* which is responsible for handling the subject-based communication. TIB/Rendezvous applies receiver-side filtering of messages. Whenever a message is published, it is sent to each host on the local network running a rendezvous daemon that delivers the message to the subscribers residing on the same host in case of a matching subscription. In case of a distributed architecture expanded over a wide-area network, *rendezvous router daemons* are used for communicating with router daemons on remote networks. Router daemons are aware of the overlay network topology and compute a multicast tree for publishing messages to other remote networks. A router daemon multicasts the messages published on its local network to remote networks, and forwards the messages coming from other remote networks. Note that the receiver-side filtering might cause significant bandwidth consumption: The overlay network can be flooded by unneeded messages because subscription information is not distributed between router daemons, and notifications might be forwarded to remote networks that host no subscribers for these messages.

3.1.4 JEDI

The Java Event-based Distributed Infrastructure (JEDI) [19, 32, 33] is a lightweight middleware infrastructure that supports the development of event-based applications. JEDI is based on the concept of active objects (AO) and event dispatchers (EDs). An AO is a special kind of object that interacts with other AOs by producing and consuming events. Thus, an AO can perform the activities of both an event publisher and a subscriber to a particular event type. ED is a special component responsible for delivering events from publishing AOs to AOs that have expressed the interest in receiving such events.

JEDI events and event patterns. JEDI event is an ordered set of attributes that describes an event characteristic. An attribute is a name-value pair: Both name and value are strings and, as a consequence, an event is a sequence of strings. JEDI supports the content-based event filtering that applies pattern matching based on regular expressions when comparing events to subscriptions. AO can either subscribe to a specific event, or to an event pattern. Event patterns are ordered set of strings that represent a simple form of regular expressions over events. An event pattern is identified with a sequence of pairs `(name, regular expression)`, where name and regular expression are both strings. A pattern-matching algorithm is used to verify compatibility between an event instance and an event pattern. For example, the event pattern `(Source_ID, 12*); (Signal_Type,*)` is compatible with all events with a value for attribute `Source_ID` starting with 12, and with any value of attribute `Signal_Type`.

ED stores all event patterns received from the subscribing AOs. When ED receives an event, it verifies compatibility between the received event and each event pattern using the pattern-matching

algorithm, and delivers the event to each AO with the matching subscription connected to it.

Distributed ED architecture. ED is a logically centralized component that needs the global knowledge about AOs, their subscriptions, and published events. However, a centralized implementation of an ED would be a critical bottleneck for a distributed system. To solve the scalability problem, JEDI offers a distributed implementation of ED. The distributed version of ED consists of a set of dispatching servers (DSs). DSs are connected to form a tree topology. Each DS is located at a different network node and is connected to one parent DS, and to zero or more descendant DSs. A DS with no parent DS is the root of the tree, while DSs with no descendant DSs are the leaves of the tree. AOs can connect to all DSs that form the ED.

DSs use a coordination protocol that distributes the information about subscriptions and events among them. The distribution protocol is designed to minimize the network load generated by control messages exchanged among the DSs. JEDI uses the hierarchical strategy to distribute events, subscriptions, and unsubscription messages between DSs: Subscriptions are propagated upwards in the tree, so that all ancestors of a DS receive it. When a DS receives a new event, it must send it to its connected AOs with a matching event pattern, to its descendant DSs that have subscribed with a matching pattern, and to its parent. This strategy ensures that all relevant nodes and the connected AOs receive the published event messages. However, this strategy has significant weakness since events are always sent upward to the root DS which may become a serious performance bottleneck.

3.1.5 Siena

Siena (Scalable Internet Event Notification Architecture) [22, 23, 24] is a middleware infrastructure that supports the implementation of publish/subscribe-based applications, with the main objective to provide a scalable Internet-scale notification service. Siena is implemented as a distributed overlay network of servers that provide clients with access points to a publish/subscribe interface. Notifications are produced by objects of interest and consumed by interested parties. Siena offers an advertisement mechanism that enables objects of interest to announce the type of notifications they intend to publish. The interested parties subscribe to notifications by defining an event pattern. Siena servers are responsible for selecting the notifications of interest and for delivering them to the interested parties.

Notifications, filters, and patterns. Siena notifications are untyped set of typed attributes. Each attribute is a triple consisting of a type, a name, and a value. A filter selects notifications by specifying attributes and constraints on the values of these attributes. Constrains are expressed by equality and ordering relations, substring, prefix, and suffix operators for strings, and the operator `any` matching any value. A filter is matched against a single notification based on the notification's attribute values. Additionally, Siena offers limited support for composite events. It is possible to investigate a combination of notifications through the use of patterns. A pattern is defined as a sequence of filters matched against a temporally ordered sequence of notifications. For example, if two notifications are received in a consecutive order, and if they match two filters that compose the client's pattern, these notifications are delivered to the client.

Distributed architecture. Siena is designed to offer scalable event distribution in wide area networks: A network of interconnected Siena servers builds the service infrastructure. Reference [24] defines and analyzes four different server topologies: centralized, hierarchical, acyclic peer-to-peer, and general peer-to-peer. A control algorithm, based on the principle of reverse path forwarding, is applied in hierarchical and peer-to-peer topologies. The main idea behind the routing algorithm is to send notifications only to the servers that have clients interested in receiving such notifications. The algorithm is based on the principles found in IP multicast:

- Downstream replication; a notification is routed in one copy as far as possible and replicated only downstream, as close as possible to the parties interested in it.
- Upstream evaluation; filers are applied and assembled upstream, as close as possible to the sources of notifications.

The forwarding of advertisements decreases the number of control messages that update subscription information, since subscription update is sent only to those servers that intend to generate matching notifications. Advertisements set the routing path for subscriptions, which in turn set the path for notifications. Every advertisement is broadcasted to all Siena servers. When a server receives a subscription, it propagates the subscription in the opposite direction along the path to the advertiser, and activates the path for notification forwarding.

3.1.6 DACs

Distributed Asynchronous Collections (DACs) are object-oriented abstractions for expressing different publish/subscribe styles and qualities of service [43, 44, 45]. A single DAC represents a distributed collection of objects, e.g., *set*, *bag*, *queue*, extended by publish/subscribe communication primitives. A client can subscribe to a DAC by registering a special *callback* object. When a new object is inserted into the DAC, it triggers the invocation of the client's listener method which delivers the inserted object to the client. Thus, DACs enable asynchronous interaction between the communicating parties.

DAC events are objects. An event can be any object with a single constraint that the object is serializable in terms of the Java programming language because serialized objects can be transported through the network. DACs offer subject-based publish/subscribe communication by using object types as the basic subscription criterion: [45] introduces the notion of *type-based subscription*. The content-based subscription scheme relies on structural reflection to implement filter objects for expressing subscription patterns [43]. Reflection offers increased modularity and flexibility compared to standard approaches that use subscription languages and patterns. However, it causes performance degradation of the filtering engine which is considerable for systems with high frequency of published events.

DACs are inherently distributed. Messages are sent to all DAC processes using various QoS such as unreliable delivery, guaranteed delivery, guaranteed delivery without duplicates, and ordered delivery [41]. A special topic membership protocol maintains the information about the running

processes and the state of channels between them. The *Topic Reliable Broadcast* protocol is defined to offer efficient and reliable message delivery resilient to channel failures.

3.1.7 Hermes

Hermes is a distributed publish/subscribe middleware system that uses peer-to-peer techniques to build and maintain a scalable overlay network of brokers for notification dissemination [89, 90]. Clients are lightweight components that can act both as notification publishers and as subscribers. They connect to system brokers that are responsible for routing notifications in the form of messages to interested subscribers. Notifications are instances of an *event type*. An event type has a type name and a list of attributes, and all event types are organized in an inheritance hierarchy.

Brokers form an overlay network that uses the principles of peer-to-peer routing algorithms, similar to the routing algorithm used in Pastry [102]. Every broker has a unique node identifier and the overlay network provides the operation of routing a message to the broker with a given node identifier. The routing of messages in the peer-to-peer layer is efficient. A message takes $O(\log N)$ hops on average to reach a destination broker, or a broker with the closest identifier, where N is the number of brokers in the system.

Hermes uses *rendezvous nodes*, similar to core nodes in the core-based multicast trees [13], for setting up delivery paths for notifications. A rendezvous node exists in the broker network for each event type. A node identifier of a rendezvous node for a given type is determined by calculating a hash value of the event type name. Hermes supports two variants of content-based routing: *type-based* and *attribute-based*. In the type-based variant, subscribers receive all notifications of a certain type and its subtypes. Attribute-based routing allows filtering of notifications according to type's attributes as close as possible to a publisher. The routing algorithm works as follows: First, advertisements that denote publisher's intent to publish notifications of a certain type are routed to the type rendezvous node. Next, subscriptions are routed to the rendezvous node to set up delivery paths for notifications. In case of the attribute-based subscriptions, subscriptions follow reverse paths of type advertisements to set up filtering state as close to the publishers as possible. Finally, a notification is routed following the advertisement message to the rendezvous node, and in the reverse direction of subscription messages from the rendezvous node. In case of attribute-based routing, notifications just follow the reverse path of subscriptions and are filtered by brokers using the information received through subscription messages.

3.1.8 REBECA

REBECA notification service [49, 78] is a content-based publish/subscribe infrastructure comprising a set of interconnected brokers that allow clients to publish notifications for interested users. Brokers are divided into two categories: *Local brokers* serve as access points for publisher and subscriber processes, while *routers* are used for forwarding messages between their neighboring brokers.

A notification in REBECA is a message that contains information about an event that has occurred.

A notification consists of a set of attributes where each attribute is a name-value pair. Notification filters are defined as boolean functions that can be applied to notifications. Filters can be either simple atomic predicates or compound filters. Simple atomic predicates contrast attributes to values using the operators, such as equality, comparison, set operators, or string operators. A compound filter is a conjunction of simple filters.

The notification service is distributed and relies on a set of routing algorithms for delivering notifications: simple routing, identity-based routing, covering-based routing, and merging-based routing. All algorithms are based on the reverse path forwarding approach and can apply advertisements to avoid subscription flooding. In simple routing, all active filters are added to the broker routing tables with the identity of the link they originate from. This approach is not optimal because the size of the routing tables grows linearly with the number of subscriptions. The straightforward improvement of the approach is to combine equal filters in routing tables, the approach used in identity-based routing. Further improvement is the covering routing strategy which considers covering among filters to decrease the size of the routing tables. Finally, the most complex approach is merging applied to create new filters that cover the existing filters [77]. REBECA working prototype has been used to evaluate and compare the listed routing strategies in [79].

Comparison of the presented publish/subscribe systems. Table 3.1 summarizes and compares the features of the presented representative publish/subscribe systems.

3.2 Mobility Support in Publish/Subscribe Systems

Most of the existing publish/subscribe systems have been designed and optimized for stationary environments where publishers and subscribers are static, and the infrastructure itself stays fixed. The mobility-related operation is dealt with at the application layer through a sequence of *subscribe-unsubscribe-subscribe* requests: A subscriber from the application layer first defines new subscriptions and unsubscribes prior to disconnecting from the publish/subscribe system. After reconnecting to the system, the subscriber needs to re-subscribe to make the system aware of its subscriptions. However, the subscriber will not receive notifications that have been published during the time of disconnection.

We have argued in [94] that the publish/subscribe middleware itself must offer the mobility support by ensuring seamless reconnection to a new broker and by preserving notifications published during disconnection. The authors in [125] agree that mobility-related issues should be addressed by the publish/subscribe middleware, and not delegated to the application layer. Some publish/subscribe systems incorporate solutions to the problem of client mobility: The common solution stores notifications published during disconnection in a special subscriber queue and delivers the notifications after subscriber reconnection. The existing solutions extent the established stationary publish/subscribe systems to cope with client mobility while keeping the infrastructure stationary [21, 48]. The position paper [59] takes an orthogonal approach: The authors analyze the requirements of mobile pub-

Table 3.1: Comparison of the presented publish/subscribe systems

	CORBA event service	**CORBA notification service**	**JMS**	**TIB/ Rendezvous**	**JEDI**	**Siena**	**DACs**	**Hermes**	**REBECA**
subscription criteria	channel-based	content-based	content-based	subject-based	content-based	content-based	type-based	type/attribute-based	content-based
notification structure	typed Any	structured messages	structured messages	set of typed attributes	set of untyped attributes	set of typed attributes	Java objects	instances of an event type	set of attributes
filtering criteria	not specified	constraint language	message selectors	not specified	regular expressions	constraint language	structural reflection	filter expressions	compound expressions
notification persistency	no	yes	yes	no	no	no	no	no	yes
architecture	not specified	not specified	centralized/ distributed implementations	distributed	distributed (hierarchical graph)	distributed (hierarchical/ acyclic graph)	distributed	distributed (general graph)	distributed (acyclic graph)
routing strategy	not specified	not specified	flooding	flooding	reverse path forwarding	reverse path forwarding	not specified	core-based trees	reverse path forwarding

lish/subscribe systems, and discuss centralized and distributed system architectures tailored to mobile environments.

Mobility support in CEA. The Cambridge Event Architecture (CEA) [10] uses a mediator which receives notifications on behalf of a subscriber during disconnections. The mediator acts as a subscriber proxy, and can register interest in subscriber's location: When the subscriber reconnects to the system, the mediator will get a notification with the new subscriber's location, and then deliver the queued messages to the subscriber. The proposed solution is indeed interesting because it relies on the publish/subscribe infrastructure itself to transmit the information about the changing subscriber locations. However, it involves a serious security risk: A malicious party could take the role of a mediator, track subscribers which jeopardizes location privacy, and deliver bogus notifications after subscriber reconnection.

Mobility support in JEDI. JEDI [33] offers two mobility-related operations: `moveIn`, and `moveOut`. A subscriber uses `moveOut` to disconnect from a broker and `moveIn` to reconnect possibly to a new broker. A client can detach from the system, serialize its current state, and later on reconnect. The old broker stores events on behalf of the subscriber during the disconnection period and transmits them to a new broker upon reconnection. The approach solves the queuing problem, however, no details regarding the handover procedure from the old to the new broker, or the change of the delivery path is given. Reference [31] proposes a rather complex solution that updates the delivery tree in a hierarchical distributed architecture: It uses a dynamic dispatching tree that has a leader responsible for subscribers with the same subscription. This solution requires a complex protocol and further analysis is needed since no evaluation study is currently available.

Mobility in Siena. The authors of Siena present a support service for mobile, wireless clients of a distributed publish/subscribe system in [21]. The mobility service enables the movement of subscribers between different access points of a publish/subscribe system. The service uses *client proxies* and a special *client library* to manage subscriptions and notifications on behalf of a subscriber, both while the subscriber is disconnected and during the handover between different access points. A client proxy runs as a special component at an access point and stores messages for a disconnected subscriber in a special queue. The client library mediates subscriptions, and initiates a `move-out` procedure: It submits subscriptions to the client proxy that subscribes using the client's subscriptions and stores incoming messages in a special buffer. The client uses the `move-in` function to reconnect to the system: It contacts the local client proxy and submits the address of the old proxy. The old and the new proxy start a special handover procedure that transfers messages from the old proxy to the new one and then to the subscriber.

The mobility service implements a special *synchronization mechanism* to avoid lost notifications. The main principle is quite simple: When transferring subscriptions from A to be active on B, the system needs to make sure that subscriptions are active on B before terminating subscriptions on A. It is possible that during the procedure both A and B will receive the same messages. The mobility service implementation permits that a subscriber receives duplicate messages.

The presented system is independent from the underlying publish/subscribe middleware: The

portability of the mobility service has been proved through an implementation on top of three different publish/subscribe systems (Siena, JMS, and Elvin). The client library wraps the target publish/subscribe API and needs to be implemented specially for each API by adding the `move-in` and `move-out` functions, and by overriding the subscribe function of the original API.

Reference [21] offers results of an experiment that proves the applicability of the implementation. The evaluation is limited since the experiment has been performed on a broker network consisting of three broker nodes, a single publisher, and a single mobile subscriber that moves only once. The experiment includes the performance evaluation if a subscriber uses a GPRS network - which has been simulated – to access the publish/subscribe service.

Mobility in REBECA. The approach taken within the project REBECA is to extend and modify the existing publish/subscribe system to support mobile and location-dependent applications [48, 125]. The mobility service aims to support two different types of mobility: *physical mobility* and *logical mobility*. Physical mobility is similar to terminal mobility: A client is physically mobile and roams between different network domains. It can disconnect from the system and later on reconnect possibly to another broker in a different network. Its subscriptions are valid and the system stores notifications published during the disconnected period. Logical mobility is related to geographical location: As a client changes its geographical position, its subscriptions dynamically change because the published information is location-dependent.

The algorithm that is developed for physical mobility is designed for a distributed network of brokers. It applies the “queuing” approach: The old broker stores notifications for a disconnected subscriber. When the subscriber connects to a new broker, it re-issues its subscriptions, but keeps no record of the old broker address. The algorithm finds the old broker by locating a broker that is at the junction of delivery paths for the new and the old broker. It is clear how this junction broker is found if simple routing is used: Each broker stores active subscriptions for all subscribers with the subscriber identifier, and since the subscription from the old broker is still active in the system, it is simple to find the junction broker leading to both the old and the new broker. The notifications stored by the old broker are routed through the junction to the new broker and delivered to the subscriber. With simple routing the routing tables can become rather large because all brokers have the knowledge about all subscriptions. Routing algorithms that use covering and merging are better suited for mobile environments where subscriptions change more often than in static scenarios. The proposed algorithm needs further extensions in case routing based on covering or merging is applied since the process of finding a broker junction is not straightforward. The designed algorithm appears to be rather complex and there are currently no evaluation results that shows it’s applicability and performance.

Mobility in Elvin. Mobility support for Elvin [111] is one of the first implementations offering mobility to subscribers in a publish/subscribe system. It enables subscriber’s nomadic mobility without modifying the original Elvin server: The proposed solution puts a proxy server between the original Elvin server and a mobile device. The central proxy server queues messages for disconnected subscribers and delivers them upon their reconnection. The presented solution implements a queuing strategy with a time-to-live expiry. A subscriber must always connect to the central proxy server

which can become a performance bottleneck and induce significant network traffic due to triangular routing if a subscriber connects to the system from another network.

JMS-based systems supporting mobility. Recently, some of the systems that implement the JMS specification offer support for mobility [107, 84, 123]. Such systems offer a lightweight JMS-compliant API for Java-enabled mobile terminals that can be used to implement JMS-based publishers and subscribers. iBus//Mobile [107] is a commercial JMS-compliant implementation. It integrates a special gateway that serves as a mediator between a JMS provider, and JMS clients. It offers support for native clients with no JMS support: Native clients can publish and receive SMS or MMS messages that are transformed into JMS messages that can interact with the JMS provider. iBus//Mobile supports TCP, UDP, HTTP, and HTTPS as transport protocols for JMS messages. JORAM [84] is an open source project that has recently published a client API called kJORAM that adjusted to J2ME devices. Pronto [123] is an academic project: It provides a JMS-compliant middleware system that supports mobility of JMS publishers and subscribers, and implements a mobile JMS API that can run on resource-limited devices. It incorporates a mobile gateway that supports JMS in wireless networks and employs SMS, or mail as transport mechanisms for native devices that do not support Java and JMS.

3.3 Related Approaches

3.3.1 Electronic Mail

Electronic mail [113] is one of the first services on the Internet for distributing messages with arbitrary content. The introduction of mailing lists provides a powerful tool for one-to-many content dissemination: Tools for creating and maintaining mailing lists facilitate users to subscribe to and unsubscribe from mailing lists automatically, and enables topic-based publish/subscribe interaction. The main disadvantage of using mail for disseminating content to large mailing lists is resource consumption: The typical mail distribution method creates a separate mail copy for each receiver from the mailing list and sends each copy separately to the receiver even if several receivers use the same mail server. This approach can cause considerable computing load and bandwidth consumption which can lead to significant delivery delays.

Contrary to the huge success and primacy on the Internet, electronic mail is currently not widely used in the mobile domain. The main reasons for its poor acceptance are bandwidth limitations and scheduled pull-style retrieval of mail messages which requires permanent network connection. Mail readers for mobile devices that apply standard Internet protocols (POP3 and IMAP) are currently available. To solve the problems related to the pull-style operation, proprietary solutions that employ push-style message retrieval are recently being deployed: Such solutions send notifications to a user's mobile terminal when a new mail message arrives at the mail server.

3.3.2 Usenet News

Usenet news [114], one of the most popular applications in computer networking, is a worldwide distributed blackboard for disseminating news. News are grouped around specific topics, called *newsgroups*, which are organized in a hierarchical topic tree similar to subject-based subscriptions in publish/subscribe systems. The basic interaction model found in Usenet news is similar to the publish/subscribe approach: A user may post an article to a newsgroup, and another user who has subscribed to this newsgroup will eventually receive a copy of the article. Users can access the articles posted to newsgroups via a news reader that manages the interaction with the news server. The news infrastructure consisting of news servers ensures the worldwide distribution of posted news. The Network News Transfer Protocol (NNTP) is used to propagate articles among news servers. The exchange of news between the news servers may be performed in either pull or push style, while a news reader receives the articles for user topics via client-side pull. The protocol offers no delivery guaranties: messages may be lost, duplicated, and delivered without order.

3.3.3 Short Message Service

Short Message Service (SMS) is a simple messaging service widely used in today's mobile networks [67, 86]. SMS transports alphanumeric messages using the *store-and forward* paradigm: Messages are temporarily stored if users cannot retrieve them at the time of message publication. A stored message is delivered to the user terminal when it reconnects to the network. SMS offers a *point-to-point service* that enables person-to-person and machine-to-person message exchange carrying at most 140 bytes of payload, either 160 7-bit characters, or 140 8-bit characters. In addition to the point-to-point communication, SMS offers the so-called *cell broadcast service* for transmitting messages to all active terminals in a cell that have subscribed to the particular information service. This feature enables the deployment of information services carrying for example weather updates and financial reports that are examples of machine-to-person SMS usage scenarios. SMS is an extremely popular messaging service, but limited by the low bandwidth communication channels.

3.3.4 Multimedia Message Service

Multimedia Message Service (MMS) is an enhanced messaging service that exploits the access to higher bandwidth in 2.5G and 3G networks [67]. MMS enables the exchange of multimedia messages carrying text, audio, and pictures in the context of person-to-person and machine-to-person scenarios. MMS supports interoperability with electronic mail which gives rise to various usage scenarios. The concept of message notification allows deferred retrieval of messages and relies on persistent network-based storage of messages: Messages can be stored persistently in the network and controlled remotely via mobile terminals. Value-added services such as weather notifications, news updates, or location-based information are typical content dissemination applications that can be deployed using MMS as a transport mechanism. Those services lack the flexibility of subscription found in publish/subscribe systems: The subscription to value-added services is static and currently

offers no means for adjusting the service to user preferences and up-to-date needs. It would be useful to extend the MMS architecture with publish/subscribe interaction principles.

3.3.5 Application-Level Multicast

Due to the lack of widespread deployment of network-level IP multicast [35], application-level multicast has emerged as an alternative solution for the efficient distribution of content to many users across wide area networks [14, 39]. The basic idea in application-level multicast is to route and replicate the content using network end-host, rather than routers. Naturally, application-level multicast solutions are less efficient than IP multicast since they may send data packets several times over the same link.

Application-level multicast systems can be regarded as distributed publish/subscribe systems offering topic-based subscription to topic, i.e., group members. The existing solutions propose different algorithms for scalable group management, and reliable message propagation through the overlay network. Systems such as SCRIBE [25], CAN [98] and Bayeux [127] build a multicast tree over a peer-to-peer network that is responsible for scalable and fault-tolerant message routing.

3.3.6 Push Systems

Push systems [57, 69] offer timely delivery of possibly large amounts of content to many subscribers in wide area networks. These systems use channels to classify the content that is published to subscribers, and the term *push service* is used to stress that the content is actively pushed to subscribers. Push systems and publish/subscribe systems are closely related: The basic interaction model is the same: Subscribers subscribe to the service and receive the published content in push style. The main difference between the two types of systems is that push systems offer services to end users, while publish/subscribe systems are middleware. Push systems offer channel-based subscription criteria to their users, while publish/subscribe systems provide flexible and expressive subscription capabilities. The extensive comparison of push systems and publish/subscribe middleware can be found in [58].

Minstrel [57] is a Java-based push system developed at the Technical University of Vienna. The main goal is to provide flexible and secure content delivery in the area of e-commerce, and to ensure system scalability. Minstrel has a distributed architecture and employs a proprietary application-layer protocol for efficient content distribution to numerous users across a wide area network. The main Minstrel components are a broadcaster and a receiver. A broadcaster is responsible for managing channels and sending information along channels. A receiver component is responsible for subscribing a user to available channels and for receiving the content. The current receiver implementation is designed for desktop computers, and the system does not support receiver mobility.

LoL@ is a prototype location-based service offering tourist information to mobile users [96]. It has been designed using the push principle for data transfer to terminals, and relies on the SIP architecture for user mobility management.

Chapter 4

Publish/Subscribe System Model

This chapter presents the mathematical model of distributed mobility-enabled publish/subscribe systems. The model reflects the observation that publish/subscribe systems exhibit the characteristics of discrete event systems, and that their behavior is guided by a sequence of events. We use set theory notation to define the model and the rules specifying valid event sequences that cause changes to system state. The main contribution of the proposed model is the introduction of proxy subscribers and proxy publishers that enable the communication and interaction between distributed publish/subscribe brokers. The approach facilitates the description of each publish/subscribe broker with a separate basic model, and the description of interactions between brokers with the publish/subscribe communication primitives. In addition to offering a formal system definition, the publish/subscribe system model enables the design of routing algorithms for disseminating notifications to subscribers in a distributed system that will be presented in Chapter 5.

The section is organized as follows: In Section 4.1 we present the basic mathematical model and extend it with mobility-related events in Section 4.2. Section 4.3 defines the distributed model which is based on the basic mathematical model, and introduces proxy publishers and proxy subscribes for modeling connections between system brokers.

4.1 Basic Mathematical Model

We present the mathematical model of a publish/subscribe system that describes the basic publish/subscribe interaction. The model involves two types of actors, publishers and subscribers, and two types of objects, notifications and subscriptions: Subscribers define subscriptions, while publishers publish notifications. Notifications matching subscriptions are delivered to subscribers. We use set theory notation to state our definitions and describe system properties following the approach used in [27] for modeling group communication systems.

4.1.1 Structural View

We define a 4-tuple $\mathcal{B} = (\mathcal{P}, \mathcal{S}, \mathcal{N}, \mathcal{M})$ comprising a set of publishers $\mathcal{P}$, and a set of subscribers $\mathcal{S}$ that interact by using notifications from the set of notifications $\mathcal{N}$, and subscriptions from the set of subscriptions $\mathcal{M}$. $\mathcal{B}$ gives the *structural view* of a publish/subscribe system and determines the boundaries of the system's state space: It defines the type and the number of entities that can exist in a publish/subscribe system.

$\mathcal{P}$ is a finite set of publishers, $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$, where $p \geq 0$ is the total number of publishers in the system. A publisher $P_i \in \mathcal{P}$ is an actor that publishes notifications from the finite set of notifications $\mathcal{N}$.

$\mathcal{S}$ is a finite set of subscribers, $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$, where $s \geq 0$ is the total number of subscribers in the system. A subscriber $S_j \in \mathcal{S}$ is an actor that defines subscriptions from the finite set of subscriptions $\mathcal{M}$. When $P_i \in \mathcal{P}$ publishes a notification $n_{ik} \in \mathcal{N}$, n_{ik} is compared to the set of S_j's active subscriptions $\{m_{j1}, m_{j2}, \ldots, m_{jl}\}$. If n_{ik} matches at least one of S_j's subscriptions, the system delivers n_{ik} to S_j. Otherwise, no action is taken.

4.1.2 Behavioral View

Publishers, subscribers, and the system interact by performing actions. The occurrence of an action is an *event* that changes a system state. Publish/subscribe systems exhibit the properties of *discrete event systems*: A discrete event systems is a dynamic system that evolves in time in response to the occurrence of events at discrete points in time [124].

We define $A = \{a_1, a_2, \ldots, a_i, \ldots\}$, a possibly infinite set of events that provides the *behavioral view* of a publish/subscribe system. An event a_i occurs at a discrete point in time $t(a_i)$. We assume that two events cannot occur simultaneously, i.e., if $t(a_i) = t(a_j) \Rightarrow a_i = a_j$. Therefore, we have an ordered sequence of events in the system where a_i occurs before a_j, iff $t(a_i) < t(a_j)$, and $i < j$.

In the basic model the following types of events can occur:

- *publish* - publisher publishes notification,
- *subscribe* - subscriber activates a subscription,
- *unsubscribe* - subscriber cancels subscription, and
- *notify* - subscriber receives notification.

Publish. Publishers perform the action of publishing notifications. We define the event *publish* as

$$pub(P_i, n_{ij}) \mid P_i \in \mathcal{P}, n_{ij} \in \mathcal{N} \tag{4.1}$$

where a publisher P_i publishes a notification n_{ij}. The publishing event adds a new notification to the set of notifications published by P_i. If we assume that each publisher can publish the same notification n_{ij} from the finite set of notifications $\mathcal{N}$ multiple times, the number of events $pub(P_i, n_{ij})$ occurring

in the system is possibly infinite. Such events occur at different points in time and are therefore not considered equal, i.e. $[a_k = pub(P_i, n_{ij})] \neq [a_l = pub(P_i, n_{ij})]$, $k \neq l$.

Published notifications. $N(P_i)$ is the set of notifications published by P_i. $N(P_i)$ is initially an empty set that is updated when P_i publishes a new notification $n_{ij} \in \mathcal{N}$. For example, the occurrence of the event $pub(P_i, n_{ij})$ adds n_{ij} to the set $N(P_i)$. We define the set of notifications published by P_i as

$$N(P_i) = \{n_{ij} \in \mathcal{N} \mid \exists a_k \in A(a_k = pub(P_i, n_{ij}))\}, \tag{4.2}$$

and the set of all published notifications as $N(\mathcal{P}) = \bigcup_{i=1}^{p} N(P_i)$.

Subscribe. Subscribers perform the action of subscribing and unsubscribing. We define the event *subscribe* as

$$sub(S_j, m_{jk}) \mid S_j \in \mathcal{S}, m_{jk} \in \mathcal{M}, m_{jk} \notin M_A(S_j) \tag{4.3}$$

where a subscriber S_j subscribes to m_{jk}. The subscribing event adds a new subscription to the set of S_j's active subscriptions $M_A(S_j)$ iff $M_A(S_j)$ does not already contain subscription m_{jk}.

Unsubscribe. We define the event *unsubscribe* as

$$unsub(S_j, m_{jk}) \mid S_j \in \mathcal{S}, m_{jk} \in M_A(S_j) \tag{4.4}$$

where S_j terminates a subscription m_{jk}. The unsubscribing event removes an existing subscription from the set of S_j's active subscriptions $M_A(S_j)$.

Active subscriptions. $M_A(S_j) \subseteq \mathcal{M}$ is the set of S_j's active subscriptions. $M_A(S_j)$ is initially an empty set that is updated each time S_j defines a new subscription $m_{jk} \in \mathcal{M}$ or cancels an existing subscription from $M_A(S_j)$. For example, the occurrence of the event $sub(S_j, m_{jk})$ adds m_{jk} to the set $M_A(S_j)$. Conversely, the occurrence of the event $unsub(S_j, m_{jk}) \mid m_{jk} \in M_A(S_j)$ removes m_{jk} from $M_A(S_j)$. We define the set of S_j's active subscriptions as

$$M_A(S_j) = \{m_{jk} \in \mathcal{M} \mid \exists a_i \in A(a_i = sub(S_j, m_{jk})), \nexists a_l \in A(a_l = unsub(S_j, m_{jk})), l > i)\}, \tag{4.5}$$

and the set of all active subscriptions as $M_A = \bigcup_{j=1}^{s} M_A(S_j)$.

Matching. We define a boolean function $match$ over the set of notifications $\mathcal{N}$ and the set of subscriptions $\mathcal{M}$ as

$$match : \mathcal{N} \times \mathcal{M} \mapsto \{true, false\}. \tag{4.6}$$

A notification n matches a subscription m iff $match(n, m)$ evaluates to $true$. We use the following simplified notation to denote that n matches m

$$n \prec m \equiv match(n, m) = true, \tag{4.7}$$

and

$$n \not\prec m \equiv match(n, m) = false \tag{4.8}$$

to denote that n does not match m.

Notify. Subscribers are notified about the publication of a notification through the event *notify* which is defined as follows

$$notify(S_j, n_{ij}) \mid S_j \in \mathcal{S}, n_{ij} \in \mathcal{N}, n_{ij} \notin N(S_j), n_{ij} \prec m_{jk} \tag{4.9}$$

where $m_{jk} \in M_A(S_j)$, i.e., the published notification matches an active subscription of S_j, and $N(S_j)$ is the set of notifications received by S_j, i.e., S_j has not previously received n_{ij}. The number of events *notify* occurring in the system is finite because $\mathcal{S}$ and $\mathcal{N}$ are finite sets.

Received notifications. $N(S_j)$ is the set of notifications received by S_j. $N(S_j)$ is initially an empty set that is updated each time S_j receives a new notification $n_{ij} \in \mathcal{N}$. For example, the occurrence of the event $notify(S_j, n_{ij})$ adds n_{ij} to the set $N(S_j)$. We define the set of notifications received by a subscriber as a consequence of an event *notify* as

$$N(S_j) = \{n_{ij} \in \mathcal{N} \mid \exists a_k \in A(a_k = notify(S_j, n_{ij}))\}\,. \tag{4.10}$$

and the set of all received notifications as $N(\mathcal{S}) = \cup_{j=1}^{s} N(S_j)$.

To summarize, in a publish/subscribe system with a defined $\mathcal{B}$, the following events can occur:

- $pub(P_i, n_{ij}), i \in 1 \ldots p, \forall n_{ij} \in \mathcal{N}$,
- $sub(S_j, m_{jk}), j \in 1 \ldots s, \forall m_{jk} \in \mathcal{M}$,
- $unsub(S_j, m_{jk}), j \in 1 \ldots s, \forall m_{jk} \in \mathcal{M}$, and
- $notify(S_j, n_{ij}), j \in 1 \ldots s, \forall n_{ij} \in \mathcal{N}$.

Some event types can possibly occur an infinite number of times, but in a different point in time which makes each event $a_i \in A$ unique.

Publish/subscribe system. A publish/subscribe system is a tuple $\mathcal{PS} = (\mathcal{B}, A)$ consisting of a 4-tuple $\mathcal{B} = (\mathcal{P}, \mathcal{S}, \mathcal{N}, \mathcal{M})$ that defines the structure of a publish/subscribe system, and a set of events A that defines system behavior.

We model the behavior of a publish/subscribe system as a sequence of events that cause changes of system states. System state is affected by the states of system publishers and subscribers. Event occurrences cause transitions between states of individual publishers and subscribers, and cause the change of the entire system state. We observe the state change of a publisher P_i through the change of $N(P_i)$, which in turn changes the set of all published notifications in the system $N(\mathcal{P})$. We observe the state change of a subscriber S_j through the change of $M_A(S_j)$ and $N(S_j)$. The change of $M_A(S_j)$ changes the set of all active subscriptions in the system M_A. The change of $N(S_j)$ changes the set of all delivered notifications in the system $N(\mathcal{S})$.

Finite automata are used to describe the behavior of discrete event systems as a sequence of discrete events causing the change of system states. We may model a publish/subscribe system as a finite state automation (Q, A, q_0, δ) consisting of a finite set of system states Q, a set of events that cause system transitions A, an initial state $q_0 \subseteq Q$, and a transition function $\delta : Q \times A \mapsto Q$. An

event a_k causes the transition from system state $q_i \in Q$ to $q_j \in Q$, which we write as $\delta(q_i, a_k) = q_j$, or $q_i \xrightarrow{a_k} q_j$.

The behavior of a publish/subscribe system $\mathcal{PS} = (\mathcal{B}, A)$ can be modeled by an automaton with a finite set of states because the sets $\mathcal{P}$, $\mathcal{S}$, $\mathcal{N}$, and $\mathcal{M}$ that determine the system state space are finite. The sequence of events in the automaton is possibly infinite. A single event trace from an automaton defines a possible valid sequence of events and state changes of the modeled publish/subscribe system. For example, a trace $\left[q_0 \xrightarrow{sub(S,m)} q_1 \xrightarrow{pub(P,n)} q_2 \xrightarrow{notify(S,n)} q_3\right]$ is a valid event trace if $n \prec m$. It comprises three events $sub(S,m), pub(P,n), notify(S,n)$ that change the system state represented by $q_0 = \left\{ \begin{array}{l} N_P(P) = \emptyset \\ M_A(S) = \emptyset \\ N(S) = \emptyset \end{array} \right\}$ to $q_3 = \left\{ \begin{array}{l} N_P(P) = \{n\} \\ M_A(S) = \{m\} \\ N(S) = \{n\} \end{array} \right\}$.

We define rules that govern the definition of valid event sequences in publish/subscribe systems. The rules define the change of a system state caused by the occurrence of an event, and the generation of a new event as a consequence of a system state change.

Rule 1.1. Publishing rule. Each *publish* event adds a notification to the set of publisher's notifications $N(P_i)$ if the notification has not previously been published by the same publisher. Formally we write this rule as

$$pub(P_i, n_{ik}) \Rightarrow N(P_i) \leftarrow N(P_i) \cup n_{ik}. \tag{4.11}$$

Rule 1.2. Subscription rule. Each *subscribe* event adds a subscription to the set of subscriber's active subscriptions $M_A(S_j)$ if the subscription has not previously been defined by the same subscriber. Formally we write this rule as

$$sub(S_j, m_{jk}) \Rightarrow M_A(S_j) \leftarrow M_A(S_j) \cup m_{jk}. \tag{4.12}$$

Rule 1.3. Unsubscription rule. Each *unsubscribe* event removes an existing subscription from the set of subscribers's active subscriptions $M_A(S_j)$. In case no such subscription exists, the set $M_A(S_j)$ does not change. Formally we can write this rule as

$$unsub(S_j, m_{jk}) \Rightarrow M_A(S_j) \leftarrow M_A(S_j) \backslash m_{jk}. \tag{4.13}$$

Rule 1.4. Delivery rule. Every *publish* event is followed by a *notify* event if the set of subscribers with an active subscription matching a published event n_{ik} is non-empty. If a subscriber has previously received n_{ik}, or in case none of the active subscriptions matches n_{ik}, no action is taken. A publishing event can possibly be followed by a number of notify events depending on the number of subscribers with an active subscription matching the published notification. Formally we write this property as

$$[pub(P_i, n_{ik}), \exists S_j, n_{ik} \notin N(S_j), \exists m_{jl} \in M_A(S_j), n_{ik} \prec m_{jl}] \Rightarrow notify(S_j, n_{ik}). \tag{4.14}$$

Rule 1.5. Notification rule. Each *notify* event adds a notification to the set of subscribers's notifications $N(S_j)$. Formally we write this rule as

$$notify(S_j, n) \Rightarrow N(S_j) \leftarrow N(S_j) \cup n. \tag{4.15}$$

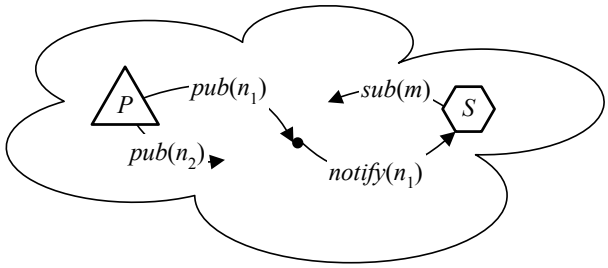

Figure 4.1: An example of a publish/subscribe system

An example of the basic model. Figure 4.1 shows an example publish/subscribe system with a single publisher and a single subscriber. $\mathcal{B} = (\{P\}, \{S\}, \{n_1, n_2\}, \{m\})$ defines the system structure. If we assume that $n_1 \prec m$ and $n_2 \not\prec m$, the list of events that can occur in the system is $pub(P, n_1)$, $pub(P, n_2)$, $sub(S, m)$, $unsub(S, m)$, and $notify(S, n_1)$.

We show the finite state automaton of the example publish/subscribe system in Figure 4.2. This is the minimal automaton of the system represented by four states. The state changes of subscriber S are observed through changes of sets $M_A(S)$ and $N(S)$. The state changes of publisher P is observed through the change of the set $N(P)$. The state changes of S are significant to state changes of the whole system, while the change of the set $N(P)$ does not significantly influence system state. The state $q_0 = \left\{ \begin{array}{l} M_A(S) = \emptyset \\ N(S) = \emptyset \end{array} \right\}$ represents the initial state in which S has not subscribed to m, and has not received any notification. When S subscribes to m, the system enters the state $q_1 = \left\{ \begin{array}{l} M_A(S) = \{m\} \\ N(S) = \emptyset \end{array} \right\}$. Event $pub(P, n_2)$ will not cause the change of system state since $n_2 \not\prec m$ and cannot be delivered to S. When in q_1, the occurrence of event $pub(P, n_1)$ will invoke another event $notify(S, n_1)$ and the system enters the state $q_2 = \left\{ \begin{array}{l} M_A(S) = \{m\} \\ N(S) = \{n_1\} \end{array} \right\}$. Further publications of n_1 or n_2 will not cause the change of system state because S has already received n_1 and can not receive it again. When S unsubscribes from m, the system enters the final state $q_3 = \left\{ \begin{array}{l} M_A(S) = \emptyset \\ N(S) = \{n_1\} \end{array} \right\}$.

4.2 Mobility-Enabled Model

We assume that either publishers or subscribers can be mobile, and the publish/subscribe service is deployed in a wired part of the network. Publishers and subscribers can disconnect from the publish/subscribe system willingly or unwillingly, and the system must accommodate such disconnections. Publisher's disconnections do not pose a significant requirement: Disconnected publishers cannot publish notifications during disconnections. The application running on publisher's terminal should store the defined notifications for further publication when it resumes the connection to the publish/subscribe service. On the other hand, subscriber's disconnections need to be supported by

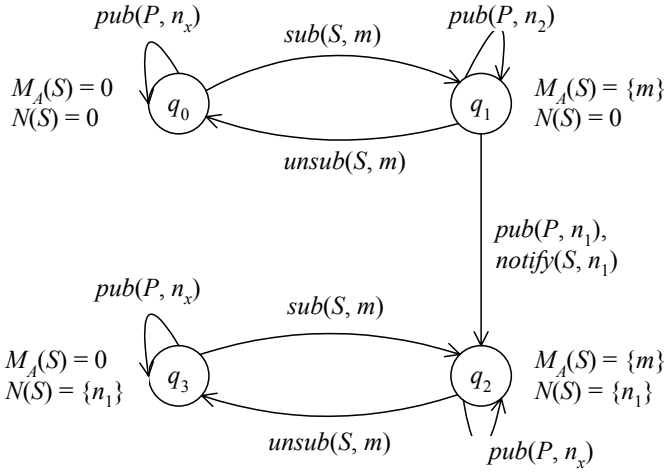

Figure 4.2: Automaton of the basic example

the publish/subscribe system: The system must enable subscribers to receive notifications that have been published during their disconnections from the system. The system must store the published notifications matching subscriber's subscription for future delivery after subscriber's reconnection to the system.

The basic mathematical model must be extended to accommodate subscriber and publisher mobility. When a subscriber is disconnected from the system, it cannot receive the published notifications matching its subscriptions because the event *notify* immediately follows *publish* as defined in eq. 4.14. We introduce *persistent notifications*: Publishers define validity periods for notifications and the system stores such notifications until their validity period expires. A disconnected subscriber can receive persistent notifications matching it's subscription that are still valid when it reconnects to the system.

We redefine the events *publish*, *subscribe*, *unsubscribe*, and *notify* defined in the basic model, and define new events, *connect*, *disconnect*, and *unpublish*, that are characteristic to the mobility-enabled model. The original event definitions for publish, subscribe, unsubscribe, and notify need to be modified since only connected publishers can publish notifications in the system, and only connected subscribers can subscribe, unsubscribe, and receive notifications in the system.

We extend the basic model with the following events:

- *connect* - a publisher or a subscriber connects to the publish/subscribe system,
- *disconnect* - a publisher or a subscriber disconnects from the publish/subscribe system, and
- *unpublish* - a publisher or the system removes a persistent notification from the system.

Connect. Publishers and subscribers can connect to the system using the event *connect*. We define the event *connect* as

$$conn(P_i) \mid P_i \in \mathcal{P}, \mathcal{P} \in B_x \tag{4.16}$$

where a publisher connects to the system described by B_x, and

$$conn(S_j) \mid S_j \in \mathcal{S}, \mathcal{S} \in B_x \tag{4.17}$$

where a subscriber connects to the system defined by B_x. The event *connect* adds a publisher to the set of connected system publishers, or a subscriber to the set of connected system subscribers.

Disconnect. Publishers and subscribers can disconnect from the system using the event *disconnect*. We define the event *disconnect* as

$$disconn(P_i) \mid P_i \in \mathcal{P}, \mathcal{P} \in B_x \tag{4.18}$$

where a publisher disconnects from the system described by B_x, and

$$disconn(S_j) \mid S_j \in \mathcal{S}, \mathcal{S} \in B_x \tag{4.19}$$

where a subscriber disconnects from the system defined by B_x. The event *disconnect* removes a publisher from the set of connected system publishers, or a subscriber from the set of connected system subscribers.

Connected publishers. $P_C \subseteq \mathcal{P}$ is the set of publishers that are connected to the system B_x. P_C is initially an empty set that is updated each time a disconnected publisher connects to the system, or when it disconnects from the system. We define the set of connected publishers as

$$P_C = \{P_i \in \mathcal{P} \mid \exists a_j \in A(a_j = conn(P_i)), \nexists a_k \in A(a_k = disconn(P_i)), j < k)\} \tag{4.20}$$

Connected subscribers. $S_C \in \mathcal{S}$ is the set of subscribers that are connected into the system. B_x. S_C is initially an empty set that is updated each time a disconnected subscriber connects to the system, or when it disconnects from the system. We define the set of connected subscribers as

$$S_C = \{S_j \in \mathcal{S} \mid \exists a_k \in A(a_k = conn(S_j)), \nexists a_l \in A(a_l = disconn(S_j)), k < l)\} \tag{4.21}$$

Only connected publishers can publish notifications. Such notifications are by default declared persistent and stored in the system until they are declared invalid, i.e., until an unpublish event causes their removal from the system. This is similar to the subscribe-unsubscribe pattern that defines and subsequently cancels active subscriptions.

Publish. We redefine the event *publish* based on the eq. 4.1 as:

$$pub(P_i, n_{ij}) \mid P_i \in P_C, n_{ij} \in \mathcal{N}, n_{ij} \notin N_P(P_i) \tag{4.22}$$

The event $pub(P_i, n_{ij})$ adds a notification n_{ij} to the set of persistent notifications published by P_i.

Unpublish. We define the event *unpublish* as

$$unpub(P_i, n_{ij}) \mid P_i \in P_C, n_{ij} \in N_P(P_i) \tag{4.23}$$

where P_i or the system removes a previously published persistent notification n_{ij} from the set of P_i's persistent notifications.

Persistent notifications. $N_P(P_i) \subseteq N(P_i)$ is the set of P_i's published persistent notifications. $N_P(P_i)$ is initially an empty set that is updated each time P_i publishes a new notification $n_{ij} \in \mathcal{N}$, or cancels a previously published notification from $N_P(P_i)$. For example, the occurrence of the event $pub(P_i, n_{ij})$ adds n_{ij} to the set $N_P(P_i)$. Conversely, the occurrence of the event $unpub(P_i, n_{ij}) \mid n_{ij} \in N_P(P_i)$ removes n_{ij} from $N_P(P_i)$. We define the set of P_i's persistent notifications as

$$N_P(P_i) = \{n_{ij} \in \mathcal{N} \mid \exists a_k \in A(a_k = pub(P_i, n_{ij})), \nexists a_l \in A(a_l = unpub(P_i, n_{ij}), k < l)\}, \tag{4.24}$$

and the set of all persistent notifications as $N_P = \bigcup_{i=1}^{p} N_P(P_i)$.

We assume that each persistent notification $n \in \mathcal{N}$ is characterized by a validity period $\Delta T(n)$ that defines the time point of notification expiry. Persistent notifications have $\Delta T(n) > 0$, and if $\Delta T(n) = 0$ the notification is non-persistent. Notifications defined in the basic mathematical model are non-persistent and become invalid immediately after their publication. Notifications in the mobility-enabled model are persistent and stored by the system until they become invalid, i.e., their validity period expires, or their publisher explicitly generates an unpublish event. A publisher P is the source of events *publish* and *unpublish*, but if it defines the validity period $\Delta T(n)$ when publishing n, then the system itself can invoke the event *unpublish* and purge invalid notification from the set of persistent notifications $N_P(P)$.

Subscribe. We redefine the event *subscribe* based on the eq. 4.3 as

$$sub(S_j, m_{jk}) \mid S_j \in S_C, m_{jk} \in \mathcal{M}, m_{jk} \notin M_A(S_j). \tag{4.25}$$

Unsubscribe. We redefine define the event *unsubscribe* based on the eq. 4.4 as

$$unsub(S_j, m_{jk}) \mid S_j \in S_C, m_{jk} \in M_A(S_j) \tag{4.26}$$

As with persistent notifications, each active subscription $m \in \mathcal{M}$ can be characterized by a validity period $\Delta T(m)$ that defines the active period of a subscription. A subscriber S defines $\Delta T(m)$ when defining a new subscription m. When $\Delta T(m)$ expires, the system can remove m from the set of active subscriptions $M_A(S)$, as if the subscriber has generated an unsubscribe event. If S generates $unsub(S, n)$ prior to m's expiry, the system removes m from the set of active subscriptions $M_A(S)$ regardless of m's validity period. The default value for subscription validity period is ∞.

Notify. We redefine define the event *notify* based on the eq. 4.9 as

$$notify(S_j, n_{ij}) \mid S_j \in S_C, n_{ij} \in N_P, n_{ij} \notin N(S_j), n_{ij} \prec m_{jk} \tag{4.27}$$

To summarize, in a publish/subscribe system that is mobility-enabled, the following events can occur:

- $conn(P_i), i \in 1 \ldots p$, $conn(S_j), j \in 1 \ldots s$,
- $disconn(P_i), i \in 1 \ldots p$, $disconn(S_j), j \in 1 \ldots s$,

- $pub(P_i, n_{ij}), P_i \in P_C, \forall n_{ij} \in \mathcal{N}$,
- $unpub(P_i, n_{ij}), P_i \in P_C, \forall n_{ij} \in \mathcal{N}$,
- $sub(S_j, m_{jk}), S_j \in S_C, \forall m_{jk} \in \mathcal{M}$,
- $unsub(S_j, m_{jk}), S_j \in S_C, \forall m_{jk} \in \mathcal{M}$, and
- $notify(S_j, n_{ij}), S_j \in S_C, \forall n_{ij} \in \mathcal{N}$.

Here we redefine the rules that are used to model the behavior of a mobility-enabled system.

Rule 2.1. Connect rule. Each event *connect* adds a publisher or a subscriber to the set of connected publishers P_C or subscribers S_C. Formally we write this rule as

$$conn(P_i) \Rightarrow P_C \leftarrow P_C \cup P_i, \tag{4.28}$$

and

$$conn(S_j) \Rightarrow S_C \leftarrow S_C \cup S_j. \tag{4.29}$$

Rule 2.2. Disconnect rule. Each event *disconnect* removes a publisher or a subscriber from the set of connected publishers P_C or subscribers S_C. Formally we write this rule as

$$disconn(P_i) \Rightarrow P_C \leftarrow P_C \backslash P_i, \tag{4.30}$$

and

$$disconn(S_j) \Rightarrow S_C \leftarrow S_C \backslash S_j. \tag{4.31}$$

Rule 2.3. Persistent publishing. Each *publish* event adds a notification to the set of publisher's persistent notifications $N_P(P_i)$ if the same notification is not already stored and valid in $N_P(P_i)$. Formally we write this rule as

$$pub(P_i, n_{ik}) \Rightarrow N_P(P_i) \leftarrow N_P(P_i) \cup n_{ik}. \tag{4.32}$$

Rule 2.4. Unpublishing rule. Each *unpublish* event removes an existing notification from the set of publishers's persistent notifications $N_P(P_i)$. If the notification is not an element of $N_P(P_i)$, the set $N_P(P_i)$ does not change. Formally we write this rule as

$$unpub(P_i, n_{ik}) \Rightarrow N_P(P_i) \leftarrow N_P(P_i) \backslash n_{ik}. \tag{4.33}$$

Rule 2.5. Subscription rule. Each *subscribe* event adds a subscription to the set of subscriber's active subscriptions $M_A(S_j)$ if the subscription has not previously been defined by the same subscriber. Formally we write this rule as

$$sub(S_j, m_{jk}) \Rightarrow M_A(S_j) \leftarrow M_A(S_j) \cup m_{jk}. \tag{4.34}$$

Rule 2.6. Unsubscription rule. Each *unsubscribe* event removes an existing subscription from the set of subscribers's active subscriptions $M_A(S_j)$. If the subscription is not an element of $M_A(S_j)$, the set $M_A(S_j)$ does not change. Formally we write this rule as

$$unsub(S_j, m_{jk}) \Rightarrow M_A(S_j) \leftarrow M_A(S_j) \backslash m_{jk}. \tag{4.35}$$

Rule 2.7. Delivery rule for connected subscribers. Every *publish* event is followed by a *notify* event if the set of connected subscribers with an active subscription matching a published event n_{ik} is non-empty. If a subscriber has previously received n_{ik}, or if none of the active subscriptions matches n_{ik}, no action is taken. Formally we write this rule as

$$[pub(P_i, n_{ik}), \exists S_j \in S_C, n_{ik} \notin N(S_j), \exists m_{jl} \in M_A(S_j), n_{ik} \prec m_{jl}] \Rightarrow notify(S_j, n_{ik}). \tag{4.36}$$

Rule 2.8. Persistent delivery after *connect*. Every event $conn(S_j)$ is possibly followed by a number of notify events if the set of persistent notifications N_P is non-empty, and if notifications from this set match any of S_j's active subscriptions. If a subscriber has previously received n_{ik}, or in case none of the active subscriptions matches n_{ik}, no action is taken. This rule ensures that subscribers receive valid notifications that have been published while they were disconnected from the system. Formally we write this rule as

$$[conn(S_j), \exists n_{ik} \in N_P, n_{ik} \notin N(S_j), \exists m_{jl} \in M_A(S_j), n_{ik} \prec m_{jl})] \Rightarrow notify(S_j, n_{ik}). \tag{4.37}$$

Rule 2.9. Persistent delivery after *subscribe*. Every event $subscribe(S_j, m_{jl})$ is possibly followed by a number of notify events if the set of persistent notifications N_P is non-empty, and if any notification from this set matches m_{jl}. If a subscriber has previously received n_{ik} no action is taken. This rule ensures that new subscribers receive valid notifications that have been published prior to definition of a new subscription. Formally we write this rule as

$$[sub(S_j, m_{jl}), S_j \in S_C, \exists n_{ik} \in N_P, n_{ik} \notin N(S_j), n_{ik} \prec m_{jk})] \Rightarrow notify(S_j, n_{ik}). \tag{4.38}$$

Rule 2.10. Notification rule. Each *notify* event adds a notification to the set of subscribers's notifications $N(S_j)$. Formally we write this rule as

$$notify(S_j, n) \Rightarrow N(S_j) \leftarrow N(S_j) \cup n. \tag{4.39}$$

An example of the mobility-enabled model. In the mobile example we analyze the behavior or a publish/subscribe system with a single publisher and a single subscriber. $\mathcal{B} = (\{P\}, \{S\}, \{n\}, \{m\})$ defines the system structure. If we assume that $n \prec m$, and that P is constantly connected to the system, the list of events that can occur in the system is $conn(S)$, $disconn(S)$, $pub(P, n)$, $sub(S, m)$, $unsub(S, m)$, and $notify(S, n)$.

The finite state automaton of the example system is shown in Figure 4.3. We observe the change of the system state through four sets: the set of connected subscribers S_C, the set of persistent notifications in the system N_P, the set of S's active subscriptions $M_A(S)$, and the set of S's received notifications $N(S)$.

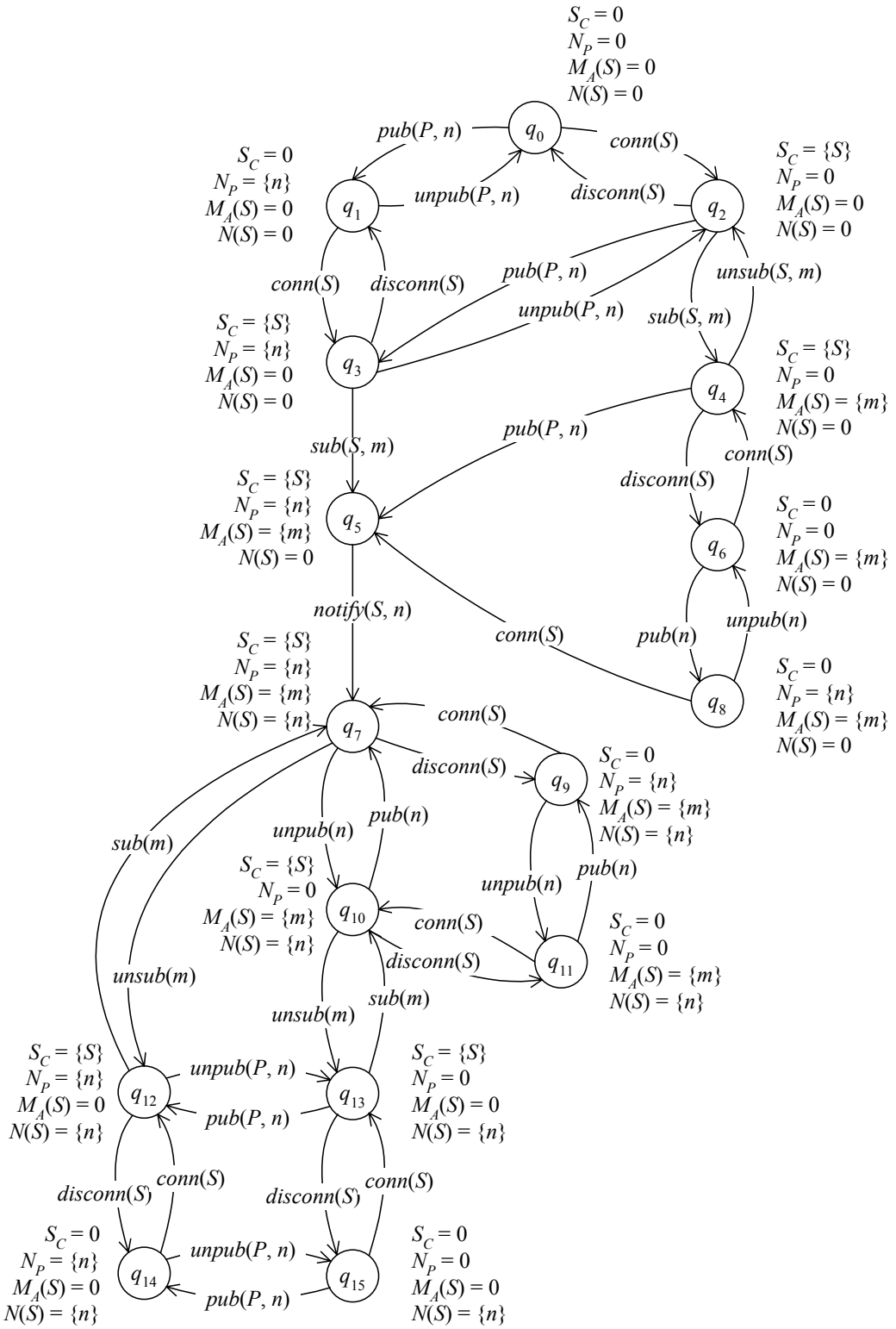

Figure 4.3: Automaton of the mobility-enabled example

In the initial state $q_0 = \left\{ \begin{array}{l} S_C = \emptyset \\ N_P = \emptyset \\ M_A(S) = \emptyset \\ N(S) = \emptyset \end{array} \right\}$, P has not published a persistent notification n, S is not connected to the system, and has not subscribed to m. When the automaton enters the state $q_5 = \left\{ \begin{array}{l} S_C = \{S\} \\ N_P = \{n\} \\ M_A(S) = \{m\} \\ N(S) = \emptyset \end{array} \right\}$, all the conditions are met to deliver n to S which causes the transition $q_5 \xrightarrow{notify(S,n)} q_7$. There are four valid event sequences that lead to q_7. The event sequence $q_0 \rightarrow q_2 \rightarrow q_4 \rightarrow q_5$ describes the most obvious event sequence leading to q_7, where S first connects to the system and subscribes to m, so that when P publishes n it can be delivered to S. The event sequence $q_0 \rightarrow q_1 \rightarrow q_3 \rightarrow q_5$ describes the situation when P first publishes n, the systems stores n in N_P, and when S connects and subscribes to m, n can be delivered to S. The sequence $q_0 \rightarrow q_2 \rightarrow q_3 \rightarrow q_5$ describes the sequence of events where S receives a previously published persistent notification n after defining a subscription m. The sequence $q_0 \rightarrow q_2 \rightarrow q_4 \rightarrow q_6 \rightarrow q_9 \rightarrow q_5$ describes the sequence of events that enable S to connect to the system, define its active subscription m, and to disconnect from the system. When P publishes n, it is stored and delivered to S after it reconnects to the system.

4.3 Distributed Model

The basic mathematical model presented in Section 4.1, and the extended model dealing with mobility presented in Section 4.2 describe publish/subscribe system as a black-box and show no details of the inner service architecture. We assume that the system has a centralized architecture dealing with all system publishers and subscribers that interact via the publish/subscribe service. However, the centralized approach has significant drawbacks with respect to scalability and fault-tolerance particularly if publishers and subscribers are scattered in a wide-area network. Therefore, it is advisory to design the service with a distributed architecture composed of a network of brokers. Each broker is a server that manages a subset of publishers and subscribers, for example, those that roam in its domain. A broker communicates with the neighboring brokers to deliver notifications from its publishers to remote subscribers residing on other brokers in the system, and to inform the neighboring brokers about subscriptions generated in its domain. The exchange of information between brokers is needed to maintain a distributed consistent view of the system as a whole.

Figure 4.4 shows an example distributed publish/subscribe system that employs three brokers for distributing notifications to subscribers. From the publisher's and subscriber's point of view, the publish/subscribe system is a black-box, while the network of brokers deals with distribution and maintains a consistent distributed system state. Assuming that publishers can publish n, and subscribers can subscribe to m, the structure of the system in Figure 4.4 is defined by $\mathcal{P} = \{P_1, P_2, P_3\}$,

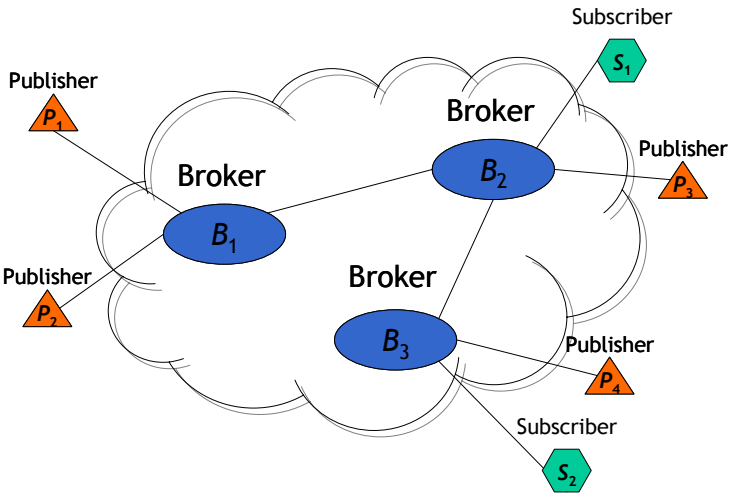

Figure 4.4: An example of a distributed publish/subscribe system

$\mathcal{S} = \{S_1, S_2\}$, $\mathcal{N} = \{n\}$, and $\mathcal{M} = \{m\}$. Each broker maintains a partial view of the system: For example, B_1 is aware of $\mathcal{P}(B_1) = \{P_1, P_2\}$ and its neighboring broker B_2. B_2 acts as a *proxy subscriber* for its local subscriber S_1, and enables B_1 to route notifications published by its local publishers P_1 and P_2 through B_2 to S_1. From the B_2's perspective, $\mathcal{P}(B_2) = \{P_3\}$ and $\mathcal{S}(B_2) = \{S_1\}$. B_2 is aware of it's neighbors B_1 and B_3, and regards B_1 as a *proxy publisher* for its local subscriber S_1. Consequently, each broker can be regarded as a publisher for the set of its local subscribers, and as a subscriber to notifications published by the set of its local publishers. We use this observation to design a distributed publish/subscribe model that includes proxy publishers and proxy subscribers that enable the communication between system brokers.

We propose a novel approach to modeling connections between brokers in a distributed publish/subscribe system. Each connection between B_i and B_j is modeled as an edge e_{ij} connecting a proxy subscriber and a proxy publisher as depicted in the upper part of Figure 4.5. It gives the model view of the system consisting of two separate publish/subscribe models $\mathcal{PS}_i$ and $\mathcal{PS}_j$. The lower part of Figure 4.5 shows the underlying system architecture, i.e., the network of brokers with connected publishers and subscribers. A proxy subscriber $S_{proxy}^{j \to i}$ represents all subscribers in B_j's domain, i.e., B_j's local subscribers and recursively remote subscribers on its neighboring brokers. $S_{proxy}^{j \to i}$, a proxy subscriber for domain j in i, is part of $\mathcal{PS}_i$ where it represents subscribers from $\mathcal{PS}_j$. A proxy publisher $P_{proxy}^{i \to j}$ represents all publishers in B_i's domain, i.e., B_i's local publishers and recursively remote publishers on its neighboring brokers. $P_{proxy}^{i \to j}$, a proxy publisher for $\mathcal{PS}_i$ in $\mathcal{PS}_j$, is part of $\mathcal{PS}_j$ where it represents publishers from $\mathcal{PS}_i$. When a subscriber from $\mathcal{PS}_j$ defines a new subscription, $S_{proxy}^{j \to i}$ must accordingly subscribe in $\mathcal{PS}_i$: $S_{proxy}^{j \to i}$ will receive notifications published in $\mathcal{PS}_i$ and forward them to the proxy publisher $P_{proxy}^{i \to j}$ that publishes notifications for subscribers in $\mathcal{PS}_j$.

The described approach enables the division of a single system model into two separate models connected by a directed edge $e_{ij} = (S_{proxy}^{j \to i}, P_{proxy}^{i \to j})$. The direction of the edge e_{ij} shows the direction

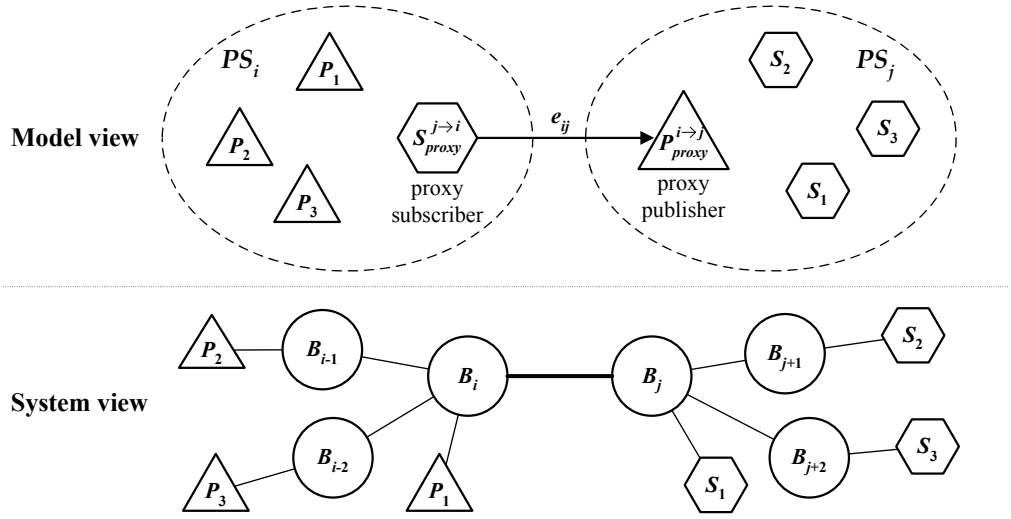

Figure 4.5: Proxy publisher and proxy subscriber

of the notification flow between $\mathcal{PS}_i$ and $\mathcal{PS}_j$. In case there are publishers in $\mathcal{PS}_j$, and subscribers in $\mathcal{PS}_i$, we need to add a proxy publisher to $\mathcal{PS}_i$, and a proxy subscriber to $\mathcal{PS}_j$. The second directed edge $e_{ji} = (S^{i \to j}_{proxy}, P^{j \to i}_{proxy})$ would connect publishers in $\mathcal{PS}_j$ with subscribers in $\mathcal{PS}_i$. The process of dividing the system model into two separate models around an existing link between two brokers can be applied recursively to all links connecting system brokers. In the end of such process, we obtain a single model per each broker that consists of local subscribers and publishers connected to the broker, and proxy subscribers and publishers representing neighboring brokers.

Figure 4.6 shows the model view of the example system depicted in Figure 4.4. It is composed of three basic models $\mathcal{PS}_1$, $\mathcal{PS}_2$, and $\mathcal{PS}_3$. Edge e_{12} corresponds to the link between brokers B_1 and B_2, and models the flow of notifications published in $\mathcal{PS}_1$ and forwarded to $\mathcal{PS}_2$. There is no edge e_{21} in the system because there are no subscribers in $\mathcal{PS}_1$. Edges e_{23} and e_{32} correspond to the link between brokers B_2 and B_3. Proxy subscriber $S^{3 \to 2}_{proxy}$ represents subscriber S_2 from $\mathcal{PS}_3$ in $\mathcal{PS}_2$. Proxy subscriber $S^{2 \to 3}_{proxy}$ represents subscriber S_1 from $\mathcal{PS}_2$ in $\mathcal{PS}_3$, and $S^{2 \to 1}_{proxy}$ represents both subscribers S_1 and S_2 in $\mathcal{PS}_1$. When publishers P_1 or P_2 publish a notification, $S^{2 \to 1}_{proxy}$ forwards it to their proxy publisher in $\mathcal{PS}_2$, $P^{1 \to 2}_{proxy}$. $P^{2 \to 3}_{proxy}$ is a proxy publisher for P_3, and also for P_1 and P_2 that publish notifications via $P^{1 \to 2}_{proxy}$. $P^{3 \to 2}_{proxy}$ is a proxy publisher for P_4 in $\mathcal{PS}_2$.

Distributed publish/subscribe model. We model a distributed publish/subscribe system as a connected directed graph $G = (V, E)$, $|V| = z$. A graph vertex represents a publish/subscribe system $v_i = \mathcal{PS}_i = (\mathcal{B}_i, A_i)$, where $\mathcal{B}_i = (\mathcal{P}_i, \mathcal{S}_i, \mathcal{N}_i, \mathcal{M}_i)$. $\mathcal{PS}_i \subseteq \mathcal{PS}$ models a part of the entire publish/subscribe system, i.e., it models publishers and subscribers that interact through a single broker. In other words, we regard each broker as a separate publish/subscribe system. Graph edges $E \subseteq \left\{ (S^{j \to i}_{proxy} \in \mathcal{PS}_i, P^{i \to j}_{proxy} \in \mathcal{PS}_j) \mid 1 \leq i \leq z, 1 \leq j \leq z \right\}$ represent directed connections between publish/subscribe systems $\mathcal{PS}_i$ and $\mathcal{PS}_j$. An edge $e_{ij} = (S^{j \to i}_{proxy}, P^{i \to j}_{proxy})$ models a commu-

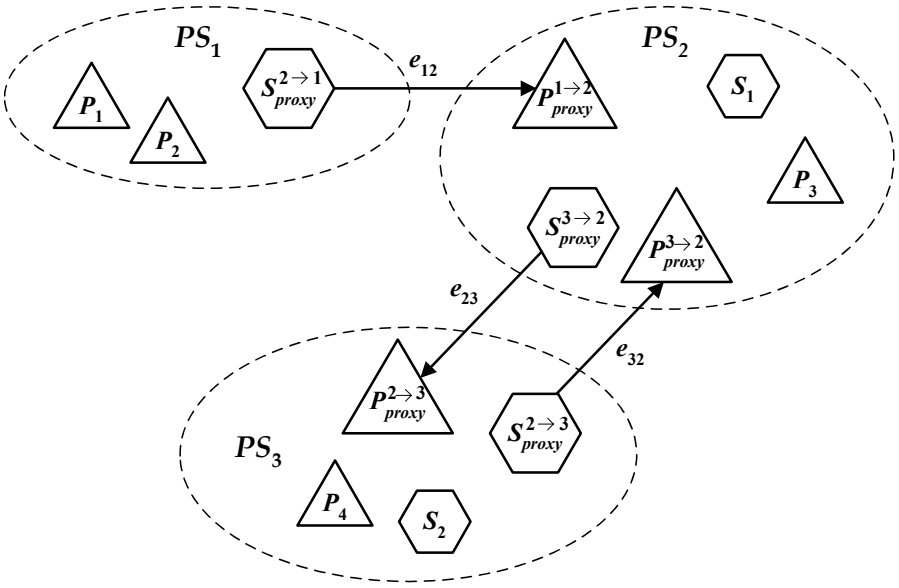

Figure 4.6: The model of the example system from Figure 4.4

nication link that enables notifications published in $\mathcal{PS}_i$ to be transmitted to $\mathcal{PS}_j$. $S^{j\rightarrow i}_{proxy} \in \mathcal{PS}_i$ is a proxy subscriber representing subscribers residing in $\mathcal{PS}_j$, while $P^{i\rightarrow j}_{proxy} \in \mathcal{PS}_j$ is a proxy publisher representing publishers from $\mathcal{PS}_i$.

The distributed model uses the mobility-enabled model presented in Section 4.2 for modeling vertices of distributed publish/subscribe system. To model a bidirectional connection between two brokers represented by $\mathcal{PS}_i$ and $\mathcal{PS}_j$, a proxy publisher $P^{j\rightarrow i}_{proxy}$ and a proxy subscriber $S^{j\rightarrow i}_{proxy}$ are created in $\mathcal{PS}_i$, and a proxy publisher $P^{i\rightarrow j}_{proxy}$ and a proxy subscriber $S^{i\rightarrow j}_{proxy}$ are created in $\mathcal{PS}_j$. When brokers disconnect, either willingly of unwillingly, proxy subscribers and proxy publishers are removed from $\mathcal{PS}_i$ and $\mathcal{PS}_j$. Consequently, it is straightforward to build a distributed system architecture consisting of a network of brokers, and to implement a simple procedure for connecting two brokers. When a broker B_i wants to connect to another broker B_j, B_i must create and connect a proxy publisher $P^{j\rightarrow i}_{proxy}$ representing the publishers of the remote broker in its own domain, and initiate the process of creating a proxy subscriber $S^{i\rightarrow j}_{proxy}$, and connecting it to the remote broker B_j. This enables B_i to forward its subscriptions to B_j and receive and republish notifications published in the remote domain. The same procedure must be performed by B_j to enable the flow of notifications from B_i to B_j.

We assume that the broker network is stationary, while publishers and subscribers are mobile entities that may change the location in the network and connect to different brokers. For example, S_2 in Figure 4.4 can disconnect from B_3 and reconnect to the system through B_1. Therefore, in a truly mobile distributed system that allows both publishers and subscribers to connect to the service via different system brokers, the following statements hold: $\mathcal{P}_i \cap \mathcal{P}_j \neq \emptyset$ and $\mathcal{S}_i \cap \mathcal{S}_j \neq \emptyset$, $1 \leq i < j \leq z$. If $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$ and $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$, $1 \leq i < j \leq z$, the system supports disconnected operation only, but

not true mobility of publishers and subscribers.

Proxy subscribers *for* $\mathcal{PS}_i$. We define the set of proxy subscribers for $\mathcal{PS}_i$ as the set of subscribers residing in neighboring publish/subscribe systems that represent subscribers from $\mathcal{PS}_i$

$$S_{out}(\mathcal{PS}_i) = \left\{ S_{proxy}^{i \to j} \mid \exists e_{ji} = (S_{proxy}^{i \to j}, P_{proxy}^{j \to i}), 1 \leq i \leq z, 1 \leq j \leq z \right\}, \tag{4.40}$$

where $S_{proxy}^{i \to j} \in \mathcal{S}_j$ and $P_{proxy}^{j \to i} \in \mathcal{P}_i$.

Proxy subscribers *in* $\mathcal{PS}_i$. We define the set of proxy subscribers residing in $\mathcal{PS}_i$ as the set of subscribers representing subscribers from neighboring publish/subscribe systems

$$S_{in}(\mathcal{PS}_i) = \left\{ S_{proxy}^{j \to i} \mid \exists e_{ij} = (S_{proxy}^{j \to i}, P_{proxy}^{i \to j}), 1 \leq i \leq z, 1 \leq j \leq z \right\}, \tag{4.41}$$

where $S_{proxy}^{j \to i} \in \mathcal{S}_i$, $P_{proxy}^{i \to j} \in \mathcal{P}_j$, and $S_{in}(\mathcal{PS}_i) \subseteq \mathcal{S}_i$.

The following statements hold: $S_{proxy}^{i \to j} \in S_{out}(\mathcal{PS}_i)$ and $S_{proxy}^{i \to j} \in S_{in}(\mathcal{PS}_j)$.

Proxy publishers *for* $\mathcal{PS}_i$. We define the set of proxy subscribers for $\mathcal{PS}_i$ as the set of publishers residing in neighboring publish/subscribe systems that represent publishers from $\mathcal{P}_i$

$$P_{out}(\mathcal{PS}_i) = \left\{ P_{proxy}^{i \to j} \mid \exists e_{ij} = (S_{proxy}^{j \to i}, P_{proxy}^{i \to j}), 1 \leq i \leq z, 1 \leq j \leq z \right\}, \tag{4.42}$$

where $P_{proxy}^{i \to j} \in \mathcal{P}_j$ and $S_{proxy}^{j \to i} \in \mathcal{S}_i$.

Proxy publishers *in* $\mathcal{PS}_i$. We define the set of proxy publishers residing in $\mathcal{PS}_i$ as the set of publishers representing publishers from neighboring publish/subscribe systems

$$P_{in}(\mathcal{PS}_i) = \left\{ P_{proxy}^{j \to i} \mid \exists e_{ji} = (S_{proxy}^{i \to j}, P_{proxy}^{j \to i}), 1 \leq i \leq z, 1 \leq j \leq z \right\}, \tag{4.43}$$

where $P_{proxy}^{j \to i} \in \mathcal{P}_i$ and $S_{proxy}^{i \to j} \in \mathcal{S}_j$.

The following statements hold: $P_{proxy}^{i \to j} \in P_{out}(\mathcal{PS}_i)$ and $P_{proxy}^{i \to j} \in P_{in}(\mathcal{PS}_j)$.

The behavior of publishers and subscribers in the distributed model follows the rules of the mobility-enabled model defined in Section 4.2. Proxy publishers and proxy subscribers follow the defined rules, and additionally, rules specific to distribution. The difference between proxy subscribers and ordinary subscribers is in the nature of generating *subscribe* and *unsubscribe* events: Ordinary subscribers generate such events at random, while proxy subscribers generate them as a consequence of a subscribe or unsubscribe event generated by one of subscribers that they represent. The similar property holds for publishers: Ordinary publishers generate publish events at random, while proxy publishers publish notifications as a consequence of the publish event generated by one of the publishers that a proxy publisher represents. We assume that each broker-related publish/subscribe system periodically removes expired persistent notifications, i.e., notifications with expired validity timestamps, from the set of N_P, without requiring a special unpublish event coming from the publisher.

Figure 4.7 illustrates the process of subscribing in a distributed model. Subscriber $S_1 \in \mathcal{S}_2$ defines a new subscription and generates the event $sub(S_1, m)$. $\mathcal{PS}_2$ forwards the subscription m to its proxy subscribers $S_{proxy}^{2 \to 1}$ and $S_{proxy}^{2 \to 3}$. Each proxy subscriber generates a new subscribe event: $sub(S_{proxy}^{2 \to 1}, m)$ adds m to $M_A(S_{proxy}^{2 \to 1})$, and $sub(S_{proxy}^{2 \to 3}, m)$ adds m to $M_A(S_{proxy}^{2 \to 3})$. In case of an

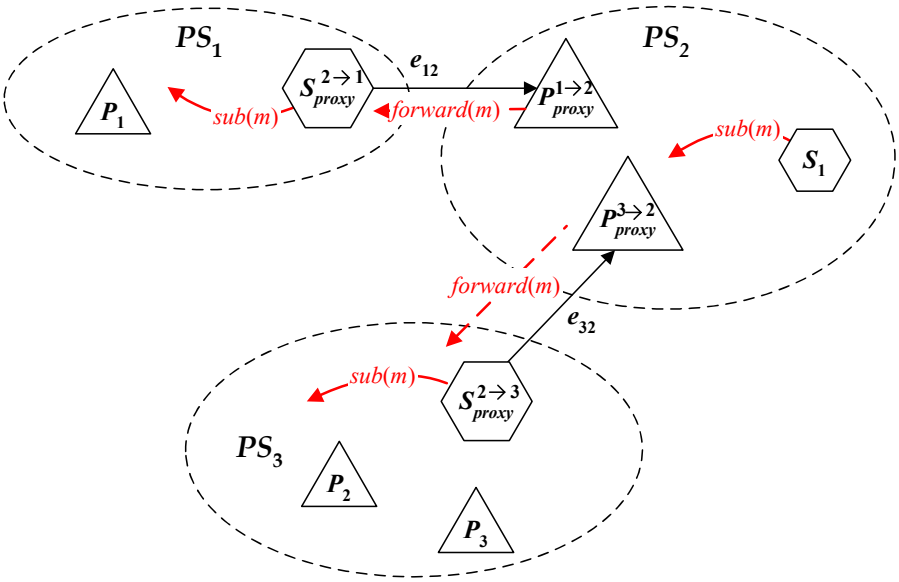

Figure 4.7: Subscribing in a distributed model

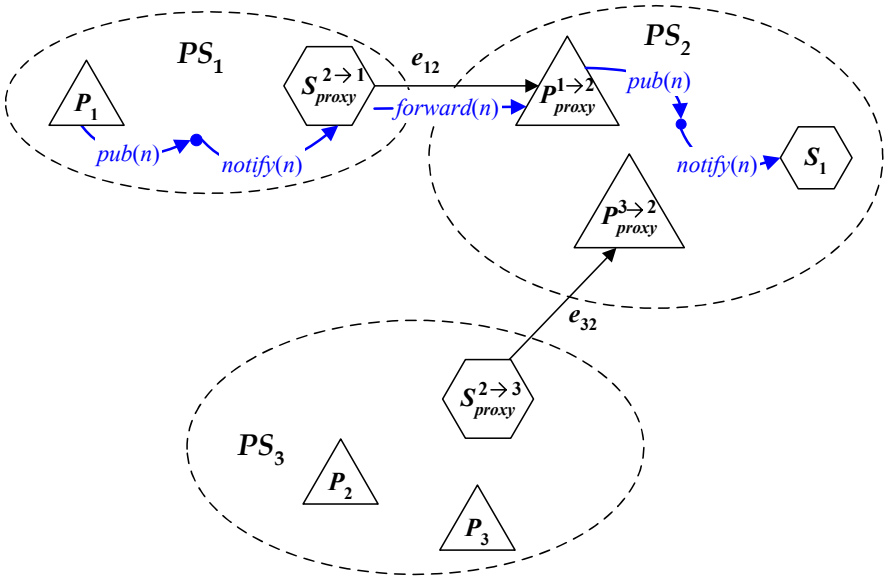

Figure 4.8: Publishing in a distributed model

unsubscribe event, $\mathcal{PS}_2$ forwards the unsubscription $\neg m$ to proxy subscribers, which causes the generation of events $unsub(S^{2\rightarrow 1}_{proxy}, m)$ and $unsub(S^{2\rightarrow 3}_{proxy}, m)$.

Figure 4.8 illustrates the process of publishing in a distributed model. When $P_1 \in \mathcal{P}_1$ publishes n, $\mathcal{PS}_1$ generates $notify(S^{2\rightarrow 1}_{proxy}, n)$ because $M_A(S^{2\rightarrow 1}_{proxy}) = \{m\}$, and $n \prec m$. When $S^{2\rightarrow 1}_{proxy}$ receives n, it forwards n to its neighboring proxy publisher $P^{1\rightarrow 2}_{proxy}$. When $P^{1\rightarrow 2}_{proxy}$ receives n, it generates a new event $pub(P^{1\rightarrow 2}_{proxy}, n)$ which invokes $notify(S_1, n)$ because $M_A(S_1) = \{m\}$ and $n \prec m$.

Here we define rules specific to the distributed environment. We assume that rules 2.1. to 2.10. defined in Section 4.2 are valid.

Rule 3.1. Forward local subscription. $\mathcal{PS}_i$ forwards a subscription generated by its local subscriber $S_{ij} \in \mathcal{S}_i \backslash S_{in}(\mathcal{PS}_i)$ to proxy subscribers in neighboring domains defined by the set $S_{out}(\mathcal{PS}_i)$. Formally we write this rule as

$$[sub(S_{ij}, m) \mid S_{ij} \in \mathcal{S}_i \backslash S_{in}(\mathcal{PS}_i)] \Rightarrow forward(S^{i\rightarrow k}_{proxy}, m), \tag{4.44}$$

where $S^{i\rightarrow k}_{proxy} \in S_{out}(\mathcal{PS}_i)$.

Rule 3.2. Forward local unsubscription. $\mathcal{PS}_i$ forwards an unsubscription generated by its local subscriber $S_{ij} \in \mathcal{S}_i \backslash S_{in}(\mathcal{PS}_i)$ to its proxy subscribers in neighboring domains defined by the set $S_{out}(\mathcal{PS}_i)$. Formally we write this rule as

$$[unsub(S_{ij}, m) \mid S_{ij} \in \mathcal{S}_i \backslash S_{in}(\mathcal{PS}_i)] \Rightarrow forward(S^{i\rightarrow k}_{proxy}, \neg m), \tag{4.45}$$

where $S^{i\rightarrow k}_{proxy} \in S_{out}(\mathcal{PS}_i)$.

Rule 3.3. Forward remote subscription. $\mathcal{PS}_i$ forwards a subscription generated by a proxy subscriber $S^{j\rightarrow i}_{proxy} \in S_{in}(\mathcal{PS}_i)$ to its proxy subscribers in neighboring domains defined by the set $S_{out}(\mathcal{PS}_i)$, except to its proxy subscriber $S^{i\rightarrow j}_{proxy} \in \mathcal{S}_j$, if such exists, because $\mathcal{PS}_j$ is the domain from which the subscription has been received. Formally we write this rule as

$$\left[sub(S^{j\rightarrow i}_{proxy}, m) \mid S^{j\rightarrow i}_{proxy} \in S_{in}(\mathcal{PS}_i)\right] \Rightarrow forward(S^{i\rightarrow k}_{proxy}, m), \tag{4.46}$$

where $S^{i\rightarrow k}_{proxy} \in S_{out}(\mathcal{PS}_i) \backslash S^{i\rightarrow j}_{proxy}$.

Rule 3.4. Forward remote unsubscription. $\mathcal{PS}_i$ forwards an unsubscription generated by a proxy subscriber $S^{j\rightarrow i}_{proxy} \in S_{in}(\mathcal{PS}_i)$ to its proxy subscribers in neighboring domains defined by the set $S_{out}(\mathcal{PS}_i)$, except to its proxy subscriber $S^{i\rightarrow j}_{proxy} \in \mathcal{S}_j$, if such exists, because $\mathcal{PS}_j$ is the domain from which the unsubscription has been received. Formally we write this rule as

$$\left[unsub(S^{j\rightarrow i}_{proxy}, m) \mid S^{j\rightarrow i}_{proxy} \in S_{in}(\mathcal{PS}_i)\right] \Rightarrow forward(S^{i\rightarrow k}_{proxy}, \neg m), \tag{4.47}$$

where $S^{i\rightarrow k}_{proxy} \in S_{out}(\mathcal{PS}_i) \backslash S^{i\rightarrow j}_{proxy}$.

Rule 3.5. Proxy subscribe. Proxy subscriber $S^{j\rightarrow i}_{proxy} \in \mathcal{S}_i$ that receives a subscription m from the remote domain subscribes to m. Formally we write this rule as

$$forward(S^{j\rightarrow i}_{proxy}, m) \Rightarrow sub(S^{j\rightarrow i}_{proxy}, m). \tag{4.48}$$

Rule 3.6. Proxy unsubscribe. Proxy subscriber $S^{j\to i}_{proxy} \in \mathcal{S}_i$ that receives an unsubscription $\neg m$ from the remote domain unsubscribes from m. Formally we write this rule as

$$forward(S^{j\to i}_{proxy}, \neg m) \Rightarrow unsub(S^{j\to i}_{proxy}, m). \tag{4.49}$$

Rule 3.7. Forward notification. Proxy subscriber $S^{j\to i}_{proxy} \in \mathcal{S}_i$ that receives a notification n forwards it using the edge e_{ij} to the proxy publisher $P^{i\to j}_{proxy} \in \mathcal{P}_j$. Formally we write this rule as

$$\Big[notify(S^{j\to i}_{proxy}, n) \mid \exists e_{ij} = (S^{j\to i}_{proxy}, P^{i\to j}_{proxy})\Big] \Rightarrow forward(P^{i\to j}_{proxy}, n). \tag{4.50}$$

Rule 3.8. Proxy publish. Proxy publisher $P^{j\to i}_{proxy} \in \mathcal{P}_i$ publishes a notification n as a consequence of receiving a notification from $S^{i\to j}_{proxy}$. Formally we write this rule as

$$forward(P^{j\to i}_{proxy}, n) \Rightarrow pub(P^{j\to i}_{proxy}, n). \tag{4.51}$$

The defined rules identify the allowed sequence of events in a distributed publish/subscribe system. It is possible to model a distributed system by a number of automata where each automaton describes a $\mathcal{PS}_i$ modeling a system broker B_i.

Chapter 5

Routing Algorithms Supporting Mobility

This chapter defines the routing algorithms for distributed publish/subscribe systems that support mobility of system publishers and subscribers. The routing algorithms are based on the mobility-enabled distributed publish/subscribe model presented in Chapter 4. We introduce a novel approach to storing notifications for disconnected users. The system stores persistent notifications until their validity period expires, and delivers such notifications to subscribers with a matching subscription as they reconnect to the system. This approach prevents the storage of notifications in special queues per each subscriber, and avoids the usage of proxy subscribers, or queues, representing disconnected subscribers in the system, which is the usual practice in the existing systems.

To prove the applicability of the proposed routing solution in a mobile setting, we have implemented a prototype system, MOPS (**Mo**bile **P**ublish **S**ubscribe), that is designed based on the defined distributed publish/subscribe model. The system is distinguishable from other publish/subscribe prototype implementations by the inherent support for publisher and subscriber mobility, as opposed to solutions that extend an existing system optimized for stationary clients. The prototype has served as an evaluation environment for assessing the performance of the proposed routing scheme based on notification persistency, and for comparing it with the standard queuing approach.

The chapter is structured as follows: Section 5.1 analyzes the existing routing algorithms for stationary systems and contrast them to the routing principles found in multicast systems. The design of routing algorithms supporting mobility based on the principle of notification persistence is presented in Section 5.2. Two different routing strategies are investigated: routing based on subscription equality (Section 5.2.1) and routing based on subscription covering (Section 5.2.2). Section 5.3 gives an evaluation of the proposed approach. A brief description of the prototype system MOPS is given in Section 5.3.1, and we give evaluation results that compare the queuing algorithm with the persistent notification algorithm in Section 5.3.2. Finally, we examine the characteristics of the proposed routing solution and discuss the evaluation results in Section 5.4.

5.1 Existing Approaches

Routing algorithms for delivering notifications to subscribers in distributed publish/subscribe systems rely on the principles found in multicast routing. Each subscription can be regarded analogous to a multicast group, where subscribers with the same active subscription represent multiple destinations that have joined a subscription-defined group. A published notification should reach each subscriber with minimal dissemination delay and network utilization: Each notification is routed in a single copy as further as possible, and multiplied when reaching a junction connecting subscriber groups along two or more network branches. The main difference between distributed publish/subscribe systems and IP multicast is in different utilization scenarios: Publish/subscribe is provided at the application layer offering sophisticated means for expressing subscriptions, while IP multicast is a network layer solution with limited subscription expressiveness, but superior to publish/subscribe systems from the point of network bandwidth consumption.

Both multicast and distributed publish/subscribe systems can be represented as graphs: Graph nodes represent routers, and graph edges physical links for IP multicast, while for publish/subscribe systems nodes model brokers and edges represent logical links between brokers. There are several routing techniques used in multicast that have influenced the design of the routing algorithms for publish/subscribe systems. These are flooding, reverse path forwarding, and core-based trees [61].

Flooding is the simplest multicast algorithm: When a network node receives a multicast packet, it ensures that it has not received it previously, and transmits a copy of the packet to all neighboring nodes, except to the one from which the packet has been received. Flooding needs to test the first reception of the packet to avoid graph loops. In case the graph is acyclic, the test in not needed.

Reverse path forwarding [35] computes a spanning tree per each multicast source using the following principle: When a multicast packet is received, test if it has arrived following the shortest path from the packet source. If this is the case, forward the packet to all neighboring nodes, except to the one from which the packet has been received. If it did not arrive through the shortest path, discard the packet. The described principle enables the forwarding of packets along a graph's spanning tree. A spanning tree is an acyclic and connected graph that connects all nodes of a given graph. The forwarding of a packet along the spanning tree will flood the network: *Pruning* is used to stop forwarding to nodes without packet recipients, i.e., pruning enables the management of group membership. Leaf nodes without packet recipients send a prune message in the reverse direction of the incoming packet. As a result of this procedure the tree will include only the nodes and edges leading to packet recipients. Some authors describe reverse path forwarding as a "flood-and-prune" algorithm.

Core-based tree algorithm [13] proposes the usage of a single tree per multicast group: A core-based tree has a core node that represents the center of a multicast group. Packet recipients send a join message to the core node to mark the path from recipients to the core node. To reach the interested recipients, each packet is first routed to the core node, and from there it follows the reverse path of join messages. The core tree approach results in a single multicast tree per group opposed to the reverse path forwarding approach which determines a different tree per each multicast source. The main

problem of the core-based tree algorithm is the choice of a core node that optimizes the dissemination delay and network load for a given group of recipients. Since the group of recipients can be dynamic, the choice of an optimal core node becomes even more challenging.

Routing algorithms in distributed publish/subscribe systems. The simplest approach that can be used for routing notifications to subscribers in publish/subscribe systems is notification flooding: Each published notification is sent to all system brokers, and brokers perform the matching of notifications to subscriptions of their local subscribers. This approach wastes a lot of bandwidth especially in cases with few or no subscribers interested in a particular published notification. Both reverse path forwarding and the core-based tree approach can be applied to disseminate the information about subscriptions in the broker network, and thus enable the routing of notifications only to interested subscribers. For example SIENA [23], JEDI [33], and REBECA [78], use the principle of reverse path forwarding in their routing protocols, while the routing algorithm applied in Hermes [89, 90] uses the core-based tree approach. We explain both approaches using an example acyclic connected graph that models nine brokers and logical links between the brokers. We use an acyclic graph to simplify the algorithm and to avoid the explanation of the procedure for determining a minimal spanning tree of a general graph. A spanning tree of an acyclic graph is equal to the graph itself. In case of a general graph, a distance-vector protocol can be applied to create a spanning tree [29].

Figure 5.1 shows the construction of a delivery tree per each publisher using reverse path forwarding. In the example network there are two publishers and two subscribers: S_1 is connected to B_1, S_2 is connected to B_6, P_1 is connected to B_9, and P_2 is connected to B_5. When a subscriber defines a new subscription, it is first submitted to the connecting broker, and further on flooded through the network of brokers. For example, when S_1 subscribes to m, this information is noted by B_1. B_1 sends a subscription request to its neighboring broker B_3, B_3 forwards it to its neighboring brokers B_2 and B_4, and so on, until the information about the subscription m reaches all brokers. When S_2 subscribes to m, this information also needs to flood the network: From S_2 it reaches B_4 through B_6. B_4 will forward it to B_3, but not to B_5 or B_7 because B_3 has previously subscribed to m at B_5 and B_7 as a consequence of S_1's subscription. Each broker maintains the information about a subscriber, or a neighboring broker that has sent a subscription to it. For example, B_1 will store a record (m, S_1) indicating that S_1 has subscribed to m. B_3 will store a record (m, B_1, B_4) indicating that it needs to send a notification matching m to both B_1 and B_4, except if the notification has not previously arrived from either B_1 and B_4. The flooding of network with subscriptions enables the definition of a delivery tree connecting all subscribers to m. The delivery tree is computed per each publisher following the reverse direction of previously flooded subscriptions. For example, the delivery tree for P_1 comprises brokers B_9, B_7, B_4, B_3, B_1, and B_6. It is computed using the local knowledge about subscriptions stored by each broker.

Figure 5.2 shows the application of a core tree for routing notifications to interested subscribers. In the example network the broker B_4 has been chosen as the core node. Each subscription is routed to the core: For example, when S_1 subscribes to m, B_1 will route it to B_4 through B_3 and create a path from B_4 to S_1 for notifications matching m. Each notification is also routed to the core which

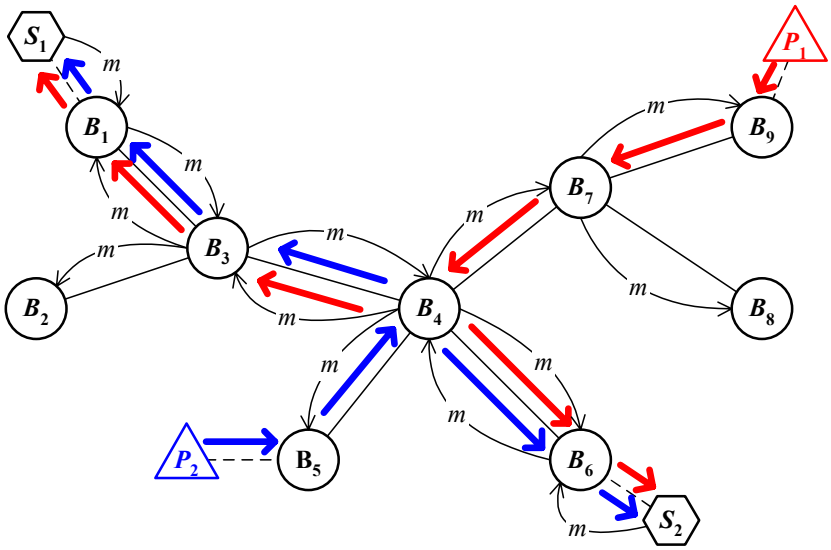

Figure 5.1: Reverse path forwarding: Creating delivery trees

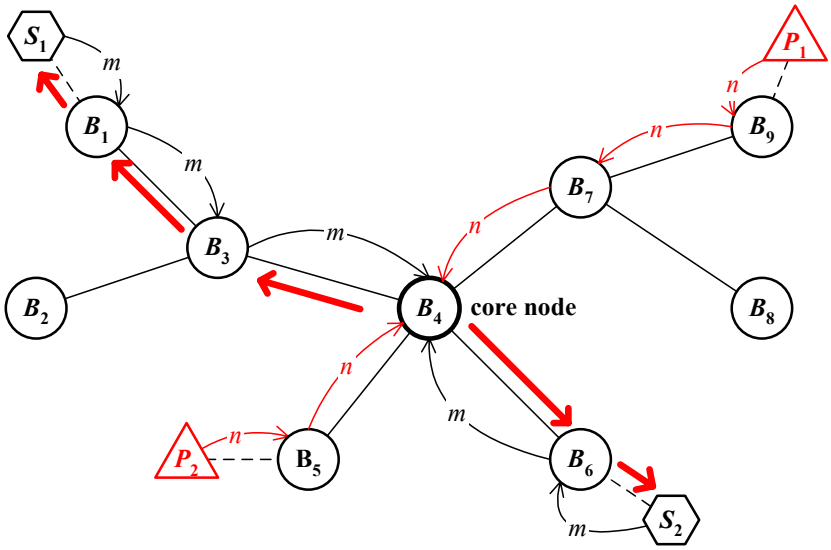

Figure 5.2: Creating a core-based tree

transmits it along the reverse path defined by subscriptions. For example, when B_2 publishes $n \prec m$, n is routed to B_4 through B_5. B_4 will duplicate the notification and send it along two paths, one of them leading to S_1, and the other to S_2. From this example it is obvious that the core-based approach is superior to the reverse path forwarding with regards to network utilization. However, the core-based approach is complex to implement: The choice of an optimal core node is proved to be an NP-complete problem. Further on, it is challenging to design an algorithm that will route each subscription or notification to the core node using only the local knowledge of each broker. Hermes [89] relies on the peer-to-peer network for routing messages to a core node. The simulation results provided in [90] prove that the core-based approach is superior in terms of network utilization and the sizes of broker's routing tables, but causes substantial notification delay when compared to the minimum spanning tree solution.

5.2 The Proposed Routing Algorithms Supporting Mobility

This section presents design of the routing algorithms for distributed publish/subscribe systems based on the distributed model presented in Section 4.3. The algorithm is specially tailored to support mobile publishers and subscribers that can connect to the publish/subscribe service using different brokers as they change the location in the network. It can be assumed that the "closest" broker, the broker residing in the same domain as a mobile publisher or subscriber, is the most suitable for providing access to the publish/subscribe service.

The common practice in publish/subscribe systems is to deliver notifications to active available subscribers as they are published. Usually, notifications are not stored by the system: It is assumed that the application using the service will provide notification storage if it is needed. However, mobile subscribers that disconnect and unsubscribe from the system, and later on reconnect and resubscribe, will not receive the notifications that have been published during the period of disconnection. Therefore, the system must store notifications on behalf of disconnected subscribers and deliver them as soon as subscribers reconnect to the system. In addition, mobile subscribers can reconnect to the system through a new broker which requires the update of the delivery path for notifications matching subscriber's subscriptions. In stationary systems the update of a delivery path occurs only when a subscriber changes its subscription, while in mobile systems such reconfigurations occur also as a consequence of subscriber's movements and reconnections through different system brokers. In mobile systems the number of control messages exchanged between system brokers is significantly increased when compared to static environments. Thus, it is vital that the routing algorithm used in mobile environments requires minimal system reconfiguration overhead added by subscriber's mobility to enable efficient, scalable, and low-delay notification delivery.

The common approach used in systems that support subscriber's mobility is to employ the last broker that has served a subscriber prior to disconnection as it's proxy: The proxy broker stores notifications in a special *subscriber's queue*. When the subscriber reconnects, possibly through another broker, a special handover procedure is performed. Firstly, the system updates the routing tables ac-

cording to the existing subscriber's subscriptions to route notifications directly to the subscriber, and secondly, the queued notifications are delivered to the new broker that delivers them to the subscriber.

The "queuing" approach requires that the system stores notifications in a special queue per each disconnected subscriber, and performs the handover procedure which is an additional overhead. Furthermore, special care must be taken that notifications are not lost or duplicated. The described approach is used for the design of a mobility service built on top of SIENA [21]: The mobility service adds proxies between system brokers and subscribers. Proxies take the role of subscribers during subscribers' disconnections. The usage of new components that act as stationary subscribers adds considerable overhead to the system as a whole. In addition, a complex procedure is performed when disconnecting and reconnecting a subscriber to the system. Another solution build on top of REBECA uses border brokers as proxies for disconnected subscribers [48]. The proposed algorithm floods the broker network with subscriptions of *all system subscribers* and requires that each broker has a copy of all subscriptions in the system which makes this approach inefficient in a system with a large number of subscriptions that often change. Furthermore, it proposes a complex procedure for modifying a delivery path from the old to a new border broker. A comprehensive performance and complexity evaluation of the proposed algorithm is needed to prove its applicability.

We propose a novel approach to deal with disconnected subscribers: It relies on the characteristic of *notification persistency* to ensure that disconnected subscribers receive the published notifications. It is important that the system stores notifications until their validity period expires which enables subscribers to receive such notifications when they reconnect to the system. If the validity period of a notification has expired, the notification is removed from the system, and disconnected users will not receive it since it is no longer relevant. A publisher, the creator of a notification, defines its validity period. Additionally, the approach enables users to receive persistent notifications after they define a new subscription, although such notifications are published prior to the definition of the new subscription.

We use *reverse path forwarding* for creating delivery trees connecting each publisher to active subscribers. When a subscriber disconnects from the system, the broker will terminate its subscriptions to invalidate the delivery path leading to the subscriber. When a subscriber reconnects, the system will re-initiate its subscriptions and create a delivery path that will enable the transmission of newly published notifications, as well as persistent notifications stored in the system. To re-initiate a subscription, a new broker needs the information about the subscriber's subscriptions, and received persistent notifications to avoid duplicate notifications. The straightforward approach is to store subscriber-related information on the subscriber's terminal. However, this solution prevents subscriber's personal mobility: A better solution is that the system stores the subscriber-related information. When a subscriber reconnects providing its credentials, the new broker first searches the system to find the information about its active subscriptions and received persistent notifications, and subsequently reactivates the subscriber's subscriptions to create a new delivery path for notifications matching the subscriptions. The system can store the information in a distributed broker network, e.g., on the last broker that served a subscriber as the access point to the publish/subscribe system. To locate the broker storing

the information, a *distributed hash table* approach can be used that finds the required information using a unique key defining the subscriber. The existing algorithms enable efficient search for data items in a distributed hash table. Data items are found by specifying a unique data key for which the algorithm finds the node storing the data associated with a given key [12]. The distributed approach offers robustness and reliability compared to the centralized solution, and seems a natural design choice for storing the information in a distributed publish/subscribe system.

The routing algorithm is based on the presented distributed model. It uses the rules 2.1 to 2.10, and 3.1 to 3.8 when defining the behavior of system brokers modeled as mobility-enabled publish/subscribe systems connected by proxy publishers and proxy subscribers. We assume that the communication in the system is remote, it is implemented by asynchronous message passing: For example, the event $notify(S, n)$ is implemented as *send message "*$notify(n)$*" to* S.

We consider two different routing strategies: routing based on subscription equality, and routing based on subscription covering. The two principles define when a broker-related system $\mathcal{PS}_x$ informs its proxy subscribers that the active subscriptions in $\mathcal{PS}_x$ have changed, and that they need to issue a new subscription or unsubscription request. Next, we define the algorithms that are independent of the applied routing strategy: These are the algorithms for building a distributed publish/subscribe environment, the algorithm enabling mobility in a distributed environment, and the algorithm for publishing notifications. The algorithms that are essential for building a distributed system are the algorithm for connecting two brokers defined in Figure 5.3, and the two algorithms describing the operation of proxy publishers and subscribers defined in Figure 5.4 and Figure 5.5, respectively. The mobility of publishers and subscribers in a distributed system is enabled by the algorithm for connecting local publishers and subscribers to a system broker, and the algorithm for disconnecting them from a system broker that are defined in Figure 5.6 and Figure 5.7, respectively. The procedure performed after the event of publishing a new notification is independent of the applied routing strategy because notifications follow delivery paths that are defined by a sequence of *subscribe* and *unsubscribe* events. The algorithm for publishing notifications is defined in Figure 5.8.

Connecting two brokers. The process of connecting two brokers modeled by $\mathcal{PS}_x$ and $\mathcal{PS}_y$ creates a communication link between the two brokers. Figure 5.3 shows the algorithm for connecting $\mathcal{PS}_y$ to $\mathcal{PS}_x$ executed by $\mathcal{PS}_x$. $\mathcal{PS}_x$ stores the list of its proxy subscribers outside $\mathcal{PS}_x$ in S_{out}, and the list of proxy subscribers and proxy publishers residing in $\mathcal{PS}_x$ in sets S_{in} and P_{in}, respectively. $\mathcal{PS}_x$ updates the list of connected publishers and subscribers using the sets P_C and S_C. Persistent notifications are stored in N_P.

Since $\mathcal{PS}_x$ wants to receive notifications coming from $\mathcal{PS}_y$, it creates a proxy publisher $P_{proxy}^{y \to x}$ that publishes notifications from $\mathcal{PS}_y$ (*line* 10), and adds the created proxy to the set P_{in} and P_C. Next, $\mathcal{PS}_x$ sends a message to $\mathcal{PS}_y$ initiating the creation of its representative proxy subscriber $S_{proxy}^{x \to y}$, and adds a reference to the proxy publisher $P_{proxy}^{y \to x}$ that forms an edge e_{yx} with $S_{proxy}^{x \to y}$ (*line* 14). $\mathcal{PS}_x$ adds a reference to $S_{proxy}^{x \to y}$ into the set S_{out}, and initiates the list of $S_{proxy}^{x \to y}$'s active subscriptions.

The method presented in lines 19 to 23 defines the sequence of actions performed when $\mathcal{PS}_x$

$S_{out} = \emptyset$ //proxy subscribers $\notin \mathcal{PS}_x$
$S_{in} = \emptyset$ //proxy subscribers $\in \mathcal{PS}_x$
$P_{in} = \emptyset$ //proxy subscribers $\in \mathcal{PS}_x$

$P_C = \emptyset$ //connected publishers
$S_C = \emptyset$ //connected subscribers
$N_P = \emptyset$ //persistent notifications

upon receiving a message "$conn(\mathcal{PS}_y)$" {
    $create(P^{y \to x}_{proxy}, \mathcal{PS}_y)$
    $P_{in} \leftarrow P_{in} \cup P^{y \to x}_{proxy}$
    $P_C \leftarrow P_C \cup P^{y \to x}_{proxy}$
    $N_P(P^{y \to x}_{proxy}) = \emptyset$
    send message "$create(S^{x \to y}_{proxy}, P^{y \to x}_{proxy})$" to $\mathcal{PS}_y$
    $S_{out} \leftarrow S_{out} \cup S^{x \to y}_{proxy}$
    $M_A(S^{x \to y}_{proxy}) = \emptyset$
}

upon receiving a message "$create(S^{y \to x}_{proxy}, P^{x \to y}_{proxy})$" {
    $create(S^{y \to x}_{proxy}, P^{x \to y}_{proxy})$
    $S_{in} \leftarrow S_{in} \cup S^{y \to x}_{proxy}$
    $S_C \leftarrow S_C \cup S^{y \to x}_{proxy}$
}

Figure 5.3: Algorithm for $\mathcal{PS}_x$: Connecting $\mathcal{PS}_y$ to $\mathcal{PS}_x$

define $P^{x \to y}_{proxy}$
upon receiving a message "$notify(n_{ik})$" from $\mathcal{PS}_x$
send message "$forward(n_{ik})$" to $P^{x \to y}_{proxy}$

upon receiving a message "$forward(m_{jl})$" from $\mathcal{PS}_y$
send message "$sub(m_{jl})$" to $\mathcal{PS}_x$

upon receiving a message "$forward(\neg m_{jl})$" from $\mathcal{PS}_y$
send message "$unsub(m_{jl})$" to $\mathcal{PS}_x$

Figure 5.4: Algorithm for the proxy subscriber $S^{y \to x}_{proxy} \in \mathcal{PS}_x$

define $S^{x \to y}_{proxy}$
upon receiving a message "$forward(n_{ik})$" from $S^{x \to y}_{proxy}$
send message "$pub(n_{ik})$" to $\mathcal{PS}_x$

Figure 5.5: Algorithm for the proxy publisher $P^{y \to x}_{proxy} \in \mathcal{PS}_x$

receives a message requesting the creation of a proxy subscriber for another broker, e.g., $\mathcal{PS}_y$. It creates a new proxy subscriber $S^{y \to x}_{proxy}$ (*line* 20) and adds a reference to the proxy publisher $P^{x \to y}_{proxy}$ through which $\mathcal{PS}_x$ publishes its notifications in $\mathcal{PS}_y$. $S^{y \to x}_{proxy}$ is added to the sets S_{in} and S_C.

Proxy subscriber $S^{y \to x}_{proxy} \in \mathcal{PS}_x$. Figure 5.4 shows the algorithm performed by a proxy subscriber residing in $\mathcal{PS}_x$ that represents subscribers from $\mathcal{PS}_y$. $S^{y \to x}_{proxy}$ stores a reference to its pair publisher in $\mathcal{PS}_y$, $P^{x \to y}_{proxy}$. When $S^{y \to x}_{proxy}$ receives a message "$notify(n_{ik})$" from $\mathcal{PS}_x$, it forwards n_{ik} to $P^{x \to y}_{proxy}$ that will publish it in $\mathcal{PS}_y$. When $S^{y \to x}_{proxy}$ receives a new subscription m_{jl} from $\mathcal{PS}_y$, it subscribes to m_{jl} in $\mathcal{PS}_x$. After receiving an unsubscription request $\neg m_{jl}$ from $\mathcal{PS}_y$, $S^{y \to x}_{proxy}$ sends the message "$unsub(m_{jl})$" to $\mathcal{PS}_x$.

Proxy publisher $P^{y \to x}_{proxy} \in \mathcal{PS}_x$. The role of a proxy publisher is quite straightforward: When it receives a notification from its pair proxy subscriber $S^{x \to y}_{proxy}$, $P^{y \to x}_{proxy}$ publishes the notification in $\mathcal{PS}_x$. The algorithm is defined in Figure 5.5.

Connecting local publishers and subscribers to $\mathcal{PS}_x$. Figure 5.6 defines the algorithm performed when a local publisher or a local subscriber connects to $\mathcal{PS}_x$. When a local publisher connects to $\mathcal{PS}_x$, it is added to the set of connected publishers P_C (*line* 2). When a local subscriber connects to $\mathcal{PS}_x$, the publisher, or the system, provides a list of its active subscriptions, and the list of persistent received notifications (*line* 4). The subscriber is added to the set of connected subscribers S_C, and if the subscriber has active subscriptions, they are reactivated: The message "$sub(S_j, m_{jl})$" for all $m_{jl} \in M_A(S_j)$ (*line* 9) subscribes S_j to m_{jl} at $\mathcal{PS}_x$ and the system creates a delivery path for notifications matching all $m_{jl} \in M_A(S_j)$ leading to the subscriber S_j. The actions following the receipt of a message "$sub(S_j, m_{jl})$" depend on the routing strategy. They are defined in Figure 5.9,

```
upon receiving a message "conn(P_i)"
    P_C ← P_C ∪ P_i

upon receiving a message "conn(S_j, M_A(S_j), N_P(S_j))" {
5       S_C ← S_C ∪ S_j
        //initiate existing subscriptions
        if M_A(S_j) ≠ ∅
            for all m_jl ∈ M_A(S_j)
                send message "sub(S_j, m_jl)" to PS_x
10  }
```

Figure 5.6: Connecting local publishers and subscribers to $\mathcal{PS}_x$

```
upon receiving a message "disconn(P_i)"
    P_C ← P_C\P_i

upon receiving a message "disconn(S_j)" {
5       S_C ← S_C\S_j
        //terminate existing subscriptions
        if M_A(S_j) ≠ ∅
            for all m_jl ∈ M_A(S_j)
                send message "unsub(S_j, m_jl)" to PS_x
10  }
```

Figure 5.7: Disconnecting local publishers and subscribers from $\mathcal{PS}_x$

Figure 5.13, and Figure 5.14.

Disconnecting local publishers and subscribers from $\mathcal{PS}_x$. Figure 5.7 defines the algorithm performed when a local publisher or a local subscriber disconnects from $\mathcal{PS}_x$. When a local publisher disconnects from $\mathcal{PS}_x$, it is removed from the set of connected publishers P_C (*line* 2). When a local subscriber disconnects from $\mathcal{PS}_x$, it is removed from the set of connected subscribers S_C. Its active subscriptions are terminated using the message "$unsub(S_j, m_{jl})$" for all $m_{jl} \in M_A(S_j)$ (*line* 9) that unsubscribes S_j from all $m_{jl} \in M_A(S_j)$ at $\mathcal{PS}_x$. The actions following the receipt of a message "$unsub(S_j, m_{jl})$" depend on the routing strategy. They are defined in Figure 5.10 and Figure 5.15.

The two algorithms for connecting and disconnecting local publishers and subscribers define the basic principle or mobility. When a subscriber disconnects, the system terminates its active subscriptions, and re-initiates them when the subscriber reconnects. The approach gives no solution to the storage of notifications published during the period of disconnection and needs to be solved elsewhere: We define the algorithm for activating new subscriptions to enable the delivery of persistent notifications stored in the system.

Notification publishing. Figure 5.8 defines the actions performed by $\mathcal{PS}_x$ when a publisher,

```
//new publication from a local or proxy publisher
upon receiving a message "pub(n_ik)" from P_i ∈ P_C
    if n_ik ∉ N_P(P_i) {
        N_P(P_i) ← N_P(P_i) ∪ n_ik
        N_P ← N_P ∪ n_ik
        for all connected local subscribers S_j ∈ [S_C \ S_in]
            if n_ik ∉ N(S_j)
                for all m_jl ∈ M_A(S_j)
                    if n_ik ≺ m_jl {
                        send message "notify(n_ik)" to S_j
                        N(S_j) ← N(S_j) ∪ n_ik
                    }

        for all connected proxy subscribers S^{y→x}_proxy ∈ S_in
            if n_ik ∉ N_P(P^{y→x}_proxy) and n_ik ∉ N(S^{y→x}_proxy)
                for all m_jl ∈ M_A(S_j)
                    if n_ik ≺ m_jl {
                        send message "notify(n_ik)" to S^{y→x}_proxy
                        N(S^{y→x}_proxy) ← N(S^{y→x}_proxy) ∪ n_ik
                    }
    }
```

Figure 5.8: Notification publishing

either a local publisher, or a proxy publisher, publishes a notification n_{ik}. $\mathcal{PS}_x$ stores the published notification in the set of persistent notifications. Next, it tests active subscriptions of all connected local subscribers, and forwards the notification to the subscriber with an active subscription matching n_{ik} in case the subscriber has not previously received the same notification (lines 6 to 12). The notification is also forwarded to connected proxy subscribers with a matching subscription using an additional test: If the notification comes from the direction of a broker the proxy subscriber represents, i.e., if it is published by $P^{y\rightarrow x}_{proxy}$, the notification is not delivered to $S^{y\rightarrow x}_{proxy}$ since the subscribers in $\mathcal{PS}_y$ have already received n_{ik} (lines 14 to 20).

5.2.1 Routing Based on Subscription Equality

The routing algorithm based on subscription equality uses the principle of *reverse path forwarding* and takes into consideration *subscription equality* when flooding the network with new subscriptions. The principle of subscription equality denotes that a subscription m originating from different subscribers in $\mathcal{PS}_x$ is considered equal by proxy subscribers $S_{out}(\mathcal{PS}_x)$ representing $\mathcal{PS}_x$'s subscribers in neighboring brokers, since m originates from the same $\mathcal{PS}_x$. In case m is currently an active subscription of one of $\mathcal{PS}_x$'s subscribers, m has already flooded the network and has set up a delivery path for notifications matching m. A new subscription event occurring in $\mathcal{PS}_x$ that defines a subscription to m by either a local subscriber, or a proxy subscriber, is not forwarded to proxy subscribers from the

set $S_{out}(\mathcal{PS}_x)$ if proxy subscribers are already subscribed to m.

Defining a new subscription. Figure 5.9 defines the actions performed by $\mathcal{PS}_x$ when a subscriber, either a local subscriber, or a proxy subscriber, defines a new subscription m_{jl}. $\mathcal{PS}_x$ checks if a subscriber is not already subscribed to m_{jl} (lines 3 and 21). If m_{jl} is not one of subscriber's active subscriptions, it is added to the set of its active subscriptions $M_A(S_j)$. m_{jl} is forwarded to $\mathcal{PS}_x$'s proxy subscribers in S_{out} if there is no active subscription to m_{jl} issued previously by one of $\mathcal{PS}_x$'s local subscribers (lines 5 and 23). Since some of the proxy subscribers may already be subscribed to m_{jl} as a consequence of a subscription by a proxy subscriber in $\mathcal{PS}_x$, we perform an additional test (*line* 7) and forward m_{jl} only to proxy subscribers without an active subscription to m_{jl}, which prevents forwarding of identical subscriptions. In case of a new subscription by a proxy subscriber, m_{jl} is forwarded to all proxy subscribers without an existing active subscription to m_{jl}, except to $S^{x\rightarrow y}_{proxy} \in \mathcal{PS}_y$, because m_{jl} has been received from $\mathcal{PS}_y$ (lines 24 to 28).

Next, the definition of a new subscription will cause the delivery of persistent notifications stored in $\mathcal{PS}_x$ that match m_{jl} and are not already in the set of persistent notifications received by the subscriber (lines 11 to 16, and 30 to 35). In case of a proxy subscriber, an additional check in needed: A proxy subscriber does not receive notifications published by $P^{y\rightarrow x}_{proxy}$ since they have already been published in $\mathcal{PS}_y$ and delivered to subscribers that $S^{y\rightarrow x}_{proxy}$ represents.

Terminating an existing subscription. Figure 5.10 defines the actions performed by $\mathcal{PS}_x$ when a subscriber, either a local subscriber, or a proxy subscriber, unsubscribes from m_{jl}. First, $\mathcal{PS}_x$ checks if a subscriber is already subscribed to m_{jl} (line 3). If the test returns $true$, m_{jl} is removed from the set of subscriber's active subscriptions (line 4). If there is no other local of remote subscriber to m_{jl} in $\mathcal{PS}_x$, the unsubscribe message is propagated to all proxy subscribers from S_{out} (lines 5 to 10). If there are no local subscribers to m_{jl} (line 12), and there are remote subscribers to m_{jl}, we count the number of proxy subscribers from S_{in} that are subscribed to m_{jl} (lines 13 to 16). In case of only one such proxy subscriber residing in $\mathcal{PS}_x$, e.g., $S^{y\rightarrow x}_{proxy}$, an unsubscription message is sent to the proxy subscriber $S^{x\rightarrow y}_{proxy}$ in $\mathcal{PS}_y$ (lines 17 to 20) because there are no other subscribers to m_{jl} in the domain of $\mathcal{PS}_x$, and a proxy subscription to m_{jl} is no longer needed. This follows the basic principle of flooding, a subscription is forwarded to all neighboring brokers, except to the one from which the subscription originates. If there are two proxy subscribers to m_{jl} in S_{in}, no subscription cancellation can be performed because all neighboring brokers need to be aware of the subscription.

5.2.2 Routing Based on Subscription Covering

The routing algorithm based on subscription covering exploits *covering* among subscriptions, the characteristic that is inherent to type-based and content-based subscriptions. The algorithm uses the principle of *reverse path forwarding* for selective forwarding of subscriptions: The covering relationship between subscriptions determines whether a new subscription should be forwarded to neighboring brokers or not. The concept of subscription covering was first defined in [22] and is denoted in literature as content-based routing. The algorithm for routing based on filter covering presented in [78]

```
//new subscription from a local subscriber
upon receiving a message "sub(m_jl)" from S_j ∈ [S_C\S_in] or "sub(m_jl, S_j)"
    if m_jl ∉ M_A(S_j) {
        M_A(S_j) ← M_A(S_j) ∪ m_jl
        if m_jl ∉ M_A(S_C\S_in) //subscription equality
            for all S_proxy ∈ S_out
                if m_jl ∉ M_A(S_proxy) {
                    send message "forward(m_jl)" to S_proxy
                    M_A(S_proxy) ← M_A(S_proxy) ∪ m_jl
                }
        if N_P ≠ ∅ //deliver persistent notifications
            for all n_ik ∈ N_P
                if n_ik ∉ N_P(S_j) and n_ik ≺ m_jl {
                    send message "notify(n_ik)" to S_j
                    N(S_j) ← N(S_j) ∪ n_ik
                }
    }

//new subscription from a proxy subscriber representing subscribers in PS_y
upon receiving a message "sub(m_jl)" from S^{y→x}_proxy ∈ S_in
    if m_jl ∉ M_A(S^{y→x}_proxy) {
        M_A(S^{y→x}_proxy) ← M_A(S^{y→x}_proxy) ∪ m_jl
        if m_jl ∉ M_A(S_C\S_in) //subscription equality
            for all S_proxy ∈ (S_out\S^{x→y}_proxy)
                if m_jl ∉ M_A(S_proxy) {
                    send message "forward(m_jl)" to S_proxy
                    M_A(S_proxy) ← M_A(S_proxy) ∪ m_jl
                }
    //deliver persistent notifications except those published in PS_y
        if N_P ≠ ∅
            for all n_ik ∈ [N_P\N_P(P^{y→x}_proxy)]
                if n_ik ≺ m_jl {
                    send message "notify(n_ik)" to S^{y→x}_proxy
                    N(S^{y→x}_proxy) ← N(S^{y→x}_proxy) ∪ n_ik
                }
    }
```

Figure 5.9: Defining a new subscription using subscription equality

```
//unsubscription from either a local or proxy subscriber
upon receiving a message unsub(m_jl) from S_j ∈ S_C or "unsub(m_jl, S_j)"
    if m_jl ∈ M_A(S_j) {
        M_A(S_j) ← M_A(S_j)\m_jl
        if m_jl ∉ M_A(S_C) //no other subscriber to m_jl
        //terminate delivery path
            for all S_proxy ∈ S_out {
                send message "forward(¬m_jl)" to S_proxy
                M_A(S_proxy) ← M_A(S_proxy)\m_jl
            }
        else
            if m_jl ∉ M_A(S_C\S_in) //no other local subscriber to m_jl {
                proxySubs = 0 //no. of proxy subscribers to m_jl
                for all S_proxy ∈ S_in
                    if m_jl ∈ M_A(S_proxy)
                        proxySubs++
                if proxySubs == 1 {
                    send message "forward(¬m_jl)" to S^{x→y}_proxy where m_jl ∈ M_A(S^{y→x}_proxy)
                    M_A(S^{x→y}_proxy) ← M_A(S^{x→y}_proxy)\m_jl
                }
            }
    }
```

Figure 5.10: Terminating an existing subscription using subscription equality

exploits the covering of subscriptions, to avoid unnecessary forwarding of subscriptions and unsubscriptions among neighboring brokers. The presented algorithm is used in stationary publish/subscribe systems. We use a similar approach in our algorithms for subscription and unsubscription covering.

The covering-based approach aims at removing covered redundant subscriptions from the system to reduce the number of subscription entries, and improve the routing performance. The main idea in covering-based routing is the following:

- A subscription request is not forwarded to neighboring brokers if a subscription covering the subscription is already active on the broker. An unsubscription request is not forwarded to neighboring brokers if a subscription covering the expired subscription is still active on the broker.

- When a subscription covering an existing subscription is defined on a broker, the broker forwards the new subscription request to its neighbors, and cancels all covered subscriptions. When the covering subscription is canceled, the broker forwards the unsubscription request for the covering subscription together with a number of subscription requests that will initiate previously covered subscriptions.

Subscription covering. A subscription m_x covers another subscription m_y if m_x matches all notifications that match m_y. We define the set of all notifications matching m_x as $N(m_x) = \{n \mid n \prec m_x\}$, and the set of all notifications matching m_y as $N(m_y) = \{n \mid n \prec m_y\}$. Formally, m_x covers m_y, denoted by $m_x \succ m_y$ iff $N(m_x) \supseteq N(m_y)$. If $m_x \succ m_y$ then $n \in N(m_y)$ implies $n \in N(m_x)$, and $n \prec m_y$ implies $n \prec m_x$.

We define the boolean function $covers$ between a set of subscriptions $\mathcal{M}$ according to the eq. 4.6 as

$$cover : \mathcal{M} \times \mathcal{M} \mapsto \{true, false\}\,. \tag{5.1}$$

Subscription covering sets. We define a subscription covering set of a proxy subscriber $S_{proxy} \in S_{out}(\mathcal{PS}_x)$ as a set of active subscriptions in $\mathcal{PS}_x$ that are covered by an existing S_{proxy}'s subscription $m_{proxy} \in M_A(S_{proxy})$. Formally,

$$M_{cov}(m_{proxy}, S_{proxy}) = \{m_{jl} \mid m_{proxy} \succ m_{jl}, m_{jl} \in M_A(S_C), m_{proxy} \in M_A(S_{proxy})\}\,. \tag{5.2}$$

We use subscription covering sets for each proxy subscriber $S_{proxy} \in S_{out}$ of a broker modeled by $\mathcal{PS}_x$ to denote covered subscriptions that are currently active in $\mathcal{PS}_x$, but that are not at the same time active for proxy subscribers $S_{proxy} \in S_{out}$. Proxy subscribers are not aware of their subscription covering sets: The broker stores and updates $M_{cov}(m_{proxy}, S_{proxy})$ for all $m_{proxy} \in M_A(S_{proxy})$. Subscription covering sets track changes to subscriptions that are not propagated to proxy subscribers: They are needed for optimization purposes introduced by subscription covering. The covering sets enable a broker to decide whether to forward a new subscription or unsubscription request to a proxy subscriber.

```
//update the S_proxy 's covering set with m_jl
updateCoveringSets(m_jl, S_proxy) {
    proxyCovered = false
    if m_jl ∈ M_A(S_proxy)
        proxyCovered = true
    else {
        for all m_proxy ∈ M_A(S_proxy)
            if m_jl ≺ m_proxy {
                proxyCovered = true; added = false
                for all m_cov ∈ M_cov(m_proxy, S_proxy)
                    if m_cov ≻ m_jl
                        added = true
                        if M_cov(m_cov, S_proxy) ≠ ∅
                            M_cov(m_cov, S_proxy) ← M_cov(m_cov, S_proxy) ∪ m_jl
                        else
                            M_cov(m_cov, S_proxy) = {m_jl}
                    else if m_jl ≻ m_cov
                        added = true
                        M_cov(m_proxy, S_proxy) ← M_cov(m_proxy, S_proxy) ∪ m_jl
                        M_cov(m_proxy, S_proxy) ← M_cov(m_proxy, S_proxy)\m_cov
                        if M_cov(m_jl, S_proxy) ≠ ∅
                            M_cov(m_jl, S_proxy) ← M_cov(m_jl, S_proxy) ∪ m_cov
                        else
                            M_cov(m_jl, S_proxy) = {m_cov}
                if added = false
                    M_cov(m_proxy, S_proxy) ← M_cov(m_proxy, S_proxy) ∪ m_jl
            }
        if proxyCovered ≡ false {
            M_cov(m_jl, S_proxy) = ∅
            for all m_proxy ∈ M_A(S_proxy)
                if m_jl ≻ m_proxy
                    M_cov(m_jl, S_proxy) ← M_cov(m_jl, S_proxy) ∪ m_proxy
        }
    }
    return proxyCovered
}
```

Figure 5.11: A method for updating a proxy subscriber's covering set with m_{jl}

```
//remove m_jl from S_proxy 's covering sets
updateCoveringSets(¬m_jl, S_proxy) {
    if M_cov(m_jl, S_proxy) ≠ ∅
        M_cov(m_jl, S_proxy) = ∅
    for all M_cov(m_proxy, S_proxy)
        if m_jl ∈ M_cov(m_proxy, S_proxy)
            M_cov(m_proxy, S_proxy) ← M_cov(m_proxy, S_proxy)\m_jl
}
```

Figure 5.12: A method for removing m_{jl} from the covering sets of a proxy subscriber

Updating a proxy subscriber's covering set with a new subscription. We define the method for updating S_{proxy}'s subscription covering set with a new subscription m_{jl} in Figure 5.11. The new subscription m_{jl} is compared to the set of active subscriptions $M_A(S_{proxy})$, and the existing subscription covering sets $M_{cov}(m_{proxy}, S_{proxy})$. We use the variable $proxyCovered$ to denote whether the new subscription m_{jl} is already covered by an active subscription from the set $M_A(S_{proxy})$, and use the value of $proxyCovered$ as the method return parameter.

If m_{jl} is already an active subscription in $M_A(S_{proxy})$, $proxyCovered$ is set to $true$ (line 5) and no updates of the covering sets are needed. Conversely, if m_{jl} is not an active subscription in $M_A(S_{proxy})$, we check whether an existing active subscription in $M_A(S_{proxy})$ already covers m_{jl} (lines 7 and 8). If an active subscription, e.g. m_{proxy}, covers m_{jl}, $proxyCovered$ is set to $true$ and we update the covering sets with m_{jl} (lines 10 to 26). Subscriptions form a hierarchy: It is not sufficient to simply add m_{jl} to the covering set of an active subscription that covers m_{jl} since m_{jl} must be put into the suitable hierarchical level of the subscription covering tree. For example, m_{jl} might be covered by an element that is already an element of the set $M_{cov}(m_{proxy}, S_{proxy})$, e.g. m_{cov}. m_{jl} should therefore become an element of m_{cov}'s coverage set to form the appropriate hierarchy (lines 11 to 16). Note that m_{cov} is not an active subscription for S_{proxy}. Lines 17 to 24 define the update of coverage sets if the new subscription m_{jl} is covered by an active subscription m_{proxy}, and when m_{jl} covers one of the elements of the m_{proxy}'s coverage set, i.e. $m_{proxy} \prec m_{jl} \prec m_{cov}$. In this case m_{jl} is put into the m_{proxy}'s coverage set, and m_{cov} becomes an element of the m_{jl}'s coverage set. If m_{jl} does not cover any of the elements from the m_{proxy}'s coverage set, and m_{jl} is not covered by any element from the same set, m_{jl} can be added to $M_{cov}(m_{proxy}, S_{proxy})$ (line 26).

If none of the active subscriptions covers m_{jl}, i.e., $proxyCovered = false$, S_{proxy} subscribes to m_{jl}, and a coverage set for m_{jl} is created (lines 28 to 33). All active subscriptions that are covered by the new subscription m_{jl} become members of the set $M_{cov}(m_{jl}, S_{proxy})$. The subscriptions from this set will be inactivated when S_{proxy} subscribes to m_{jl}.

Removing a subscription from proxy subscriber's covering sets. Figure 5.12 defines the algorithm for removing m_{jl} from the proxy subscriber's covering sets. Firstly, the elements from the set $M_{cov}(m_{jl}, S_{proxy})$ are removed, and secondly, m_{jl} is removed from all other S_{proxy}'s covering sets.

```
//new subscription from a local subscriber
upon receiving a message "sub(m_jl)" from S_j ∈ [S_C\S_in] or "sub(m_jl, S_j)"
    if m_jl ∉ M_A(S_j) {
        M_A(S_j) ← M_A(S_j) ∪ m_jl
        covered = false
        if m_jl ∉ M_A(S_C\S_in) {
            for all m_local ∈ M_A(S_C\S_in)
                if m_jl ≺ m_local {
                    covered = true //covered by local subscription
                    for all S_proxy ∈ S_out
                        updateCoveringSets(m_jl, S_proxy)
                }
            if covered = false
                for all S_proxy ∈ S_out {
                    proxyCovered = updateCoveringSets(m_jl, S_proxy)
                    if proxyCovered ≡ false {
                    //not covered by proxy subscriptions
                        send message "forward(m_jl)" to S_proxy
                        M_A(S_proxy) ← M_A(S_proxy) ∪ m_jl
                        for all m_cov ∈ M_cov(m_jl, S_proxy) {
                            send message "forward(¬m_cov)" to S_proxy
                            M_A(S_proxy) ← M_A(S_proxy)\m_cov
                        }
                    }
                }
        } else
            covered = true //equal to a local subscription
        if N_P ≠ ∅ //receive persistent notifications
            for all n_ik ∈ N_P
                if n_ik ∉ N_P(S_j) and n_ik ≺ m_jl {
                    send message "notify(n_ik)" to S_j
                    N(S_j) ← N(S_j) ∪ n_ik
                }
    }
```

Figure 5.13: Local subscription based on covering

Defining a new subscription by a local subscriber. Figure 5.13 defines the algorithm performed when a local subscriber defines a new subscription m_{jl}. If S_j is not already subscribed to m_{jl}, m_{jl} is added to the set of S_j's active subscriptions (line 4), and, if needed, the new subscription is propagated to proxy subscribers from the set S_{out}. In case m_{jl} is already an active subscription issued by another local subscriber, there is no need for subscription propagation (line 27). Conversely, we check whether m_{jl} is covered by an existing active subscription of a local subscriber (lines 7 and 8). If a covering subscription is active, there is no need to propagate m_{jl} to proxy subscribers. We only update the covering sets of proxy subscribers by calling the method $updateCoveringSets(m_{jl}, S_{proxy})$ (lines 10 and 11). In case m_{jl} is not covered by a local subscription, we check whether each proxy subscriber from the set S_{out} already has an active subscription covering m_{jl}. Only if such subscription is not active for a $S_{proxy} \in S_{out}$, a subscription request for m_{jl} is propagated to S_{proxy} (lines 18 and 19). We also cancel subscriptions covered by m_{jl} from the set $M_{cov}(m_{jl}, S_{proxy})$ by sending unsubscription requests to S_{proxy} (lines 20 to 23). Lines 28 to 33 ensure that a local subscriber receives persistent notifications as in the case of subscription based on equality (Figure 5.9).

Defining a new subscription by a proxy subscriber. The algorithm for defining a new subscription by a proxy subscriber as defined in Figure 5.14 is similar to the algorithm for a local subscriber. The only difference between the two algorithms is in the fact that a subscription by a proxy subscriber $S_{proxy}^{y \to x}$ is not forwarded to the proxy subscriber $S_{proxy}^{x \to y}$ since it originated from $\mathcal{PS}_y$, while a new subscription by a local subscriber is forwarded to all proxy subscribers from the set S_{out}.

Terminating an existing subscription. We define the algorithm for unsubscribing a local or a proxy subscriber from m_{jl} in Figure 5.15. The algorithm relies on the information in the covering sets that is maintained by a broker. If m_{jl} is S_j's active subscription, it is first removed from the set of active subscriptions. In case there are no other subscribers to m_{jl} connected to the broker, either local or proxy subscribers, the subscription to m_{jl} can be canceled (lines 5 to 18). The unsubscription is propagated to a proxy subscriber from the set S_{out} only in case m_{jl} is also an active subscription of the proxy subscriber $S_{proxy} \in S_{out}$ (line 8). Otherwise, no action is taken because m_{jl} is covered by another subscription and S_{proxy} should not change its subscriptions. To cancel an active subscription of a proxy subscriber S_{proxy}, we first remove m_{jl} from the set of S_{proxy}'s active subscriptions (line 9), next we initiate subscriptions to all covered subscriptions from the set $M_{cov}(m_{jl}, S_{proxy})$ (lines 10 to 14), and finally, we cancel the active subscription to m_{jl} by sending a message "$forward(\neg m_{jl})$" to S_{proxy} (line 15). The broker updates the covering sets for each proxy subscriber by invoking the method $updateCoveringSets(\neg m_{jl}, S_{proxy})$.

If there are still active subscribers to m_{jl} connected to the broker, we check whether there is a single proxy subscriber, e.g., $S_{proxy}^{y \to x}$, with an active subscription to m_{jl} connected to the broker (lines 21 to 24). In this case the active subscription by it's pair proxy subscriber $S_{proxy}^{x \to y}$ is canceled (lines 25 to 34). In all other cases, i.e., if there is a local subscriber to m_{jl}, or two proxy subscribers to m_{jl}, no actions are needed.

```
//new subscription from a proxy subscriber
upon receiving a message "sub(m_jl)" from S^{y→x}_{proxy} ∈ S_in
    if m_jl ∉ M_A(S_j) {
        M_A(S_j) ← M_A(S_j) ∪ m_jl
        covered = false
        if m_jl ∉ M_A(S_C \ S_in) {
            for all m_local ∈ M_A(S_C \ S_in)
                if m_jl ≺ m_local {
                    covered = true //covered by local subscription
                    for all S_proxy ∈ [S_out \ S^{x→y}_{proxy}]
                        updateCoveringSets(m_jl, S_proxy)
                }
            if covered = false
                for all S_proxy ∈ [S_out \ S^{x→y}_{proxy}] {
                    proxyCovered = updateCoveringSets(m_jl, S_proxy)
                    if proxyCovered ≡ false {
                    //not covered by proxy subscriptions
                        send message "forward(m_jl)" to S_proxy
                        for all m_cov ∈ M_cov(m_jl, S_proxy) {
                            send message "forward(¬m_cov)" to S_proxy
                            M_A(S_proxy) ← M_A(S_proxy) \ m_cov
                        }
                        M_A(S_proxy) ← M_A(S_proxy) ∪ m_jl
                    }
                }
        } else
            covered = true //equal to a local subscription
    //receive persistent notifications except those published in PS_y
        if N_P ≠ ∅
            for all n_ik ∈ [N_P \ N_P(P^{y→x}_{proxy})]
                if n_ik ≺ m_jl {
                    send message "notify(n_ik)" to S^{y→x}_{proxy}
                    N(S^{y→x}_{proxy}) ← N(S^{y→x}_{proxy}) ∪ n_ik
                }
    }
```

Figure 5.14: Proxy subscription based on covering

```
//unsubscription from either a local or proxy subscriber
upon receiving a message unsub(m_jl) from S_j ∈ S_C or "unsub(m_jl, S_j)"
    if m_jl ∈ M_A(S_j) {
        M_A(S_j) ← M_A(S_j)\m_jl
        if m_jl ∉ M_A(S_C) //no other subscriber to m_jl
        //terminate delivery path
            for all S_proxy ∈ S_out {
                if m_jl ∈ M_A(S_proxy) {
                    M_A(S_proxy) ← M_A(S_proxy)\m_jl
                    if M_cov(m_jl, S_proxy) ≠ ∅
                        for all m_cov ∈ M_cov(m_jl, S_proxy) {
                            send message "forward(m_cov)" to S_proxy
                            M_A(S_proxy) ← M_A(S_proxy) ∪ m_cov
                        }
                    send message "forward(¬m_jl)" to S_proxy
                }
                updateCoveringSets(¬m_jl, S_proxy)
            }
        else
            if m_jl ∉ M_A(S_C\S_in) //no other local subscriber to m_jl {
                proxySubs = 0 //no. of proxy subscribers to m_jl
                for all S_proxy ∈ S_in
                    if m_jl ∈ M_A(S_proxy)
                        proxySubs++
                if proxySubs == 1 {
                    M_A(S^{x→y}_proxy) ← M_A(S^{x→y}_proxy)\m_jl
                    if M_cov(m_jl, S^{x→y}_proxy) ≠ ∅
                        for all m_cov ∈ M_cov(m_jl, S^{x→y}_proxy) {
                            send message "forward(m_cov)" to S^{x→y}_proxy
                            M_A(S^{x→y}_proxy) ← M_A(S^{x→y}_proxy) ∪ m_cov
                        }
                    send message "forward(¬m_jl)" to S^{x→y}_proxy where m_jl ∈ M_A(S^{y→x}_proxy)
                    updateCoveringSets(¬m_jl, S^{x→y}_proxy)
                }
            }
    }
```

Figure 5.15: Terminating an existing subscription based on covering

5.3 Evaluation of the Routing Algorithms

We use the implementation of the prototype system MOPS (**Mo**bile **P**ublish **S**ubscribe) to investigate the applicability of the proposed routing algorithms in mobile scenarios. The implementation shows that mobile subscribers receive notifications published during the period of their disconnection that are still valid when subscribers reconnect to the system, possibly through another broker. Subscribers do not receive duplicate notifications because of the mechanism that compares already received notifications that are still valid to those that should be sent to a subscriber. Undelivered notifications are possible because of the delay introduced by a broker network and subscriber mobility which can lead to notification expiry prior to it's delivery to the subscriber. We evaluate the performance of the routing algorithm based on subscription covering with persistent notifications (PN-alg), and compare it to the same algorithm that uses queues (Q-alg). We define the metrics to assess the performance of publish/subscribe systems in mobile scenarios.

5.3.1 The Prototype System MOPS

MOPS has been designed and implemented to prove the concept, and evaluate the proposed publish/subscribe distributed model, and routing algorithms supporting client mobility. The system has been implemented in the Java programming language. It is distinguishable from other publish/subscribe prototype implementations by the inherent support for publisher and subscriber mobility that has been integrated into the system design, rather than added as an extension to an existing system supporting stationary clients. Furthermore, the current system implementation can be configured to use either queues for storing notifications on behalf of disconnected subscribers, or persistent notifications that are maintained by the brokers until their validity period expires.

The MOPS infrastructure comprises a set of interconnected brokers that form an acyclic communication graph. There is a single spanning tree for notification delivery connecting a publisher to a group of subscribers, and each broker is a single point of system failure. The system currently does not provide mechanisms for fault tolerance: We assume that brokers cannot fail and that the communication links between brokers are error-free bidirectional point-to-point links. The broker network is built by incrementally connecting a new broker to an active broker, and can be extended during system operation. Clients, i.e., publishers and subscribers, are mobile entities that can connect to different brokers. The communication between a pair of brokers, and a client and a broker is implemented in the form of messages that are serialized Java objects transported using TCP.

The MOPS system supports typed notifications that carry a list of attributes. It offers type-based and attribute-based subscriptions, and implements type-based routing with support for subscription covering in the broker network. Notification filtering according to attribute-based subscriptions is performed only on the edge broker prior to notification delivery to subscribers. Publishers can publish notifications of an already defined type that is defined in the broker network, and subscribers can only subscribe to recognized types. The set of recognized types is defined prior to system startup using a file containing a serialized list of Java classes, and can be updated during system operation. The file is

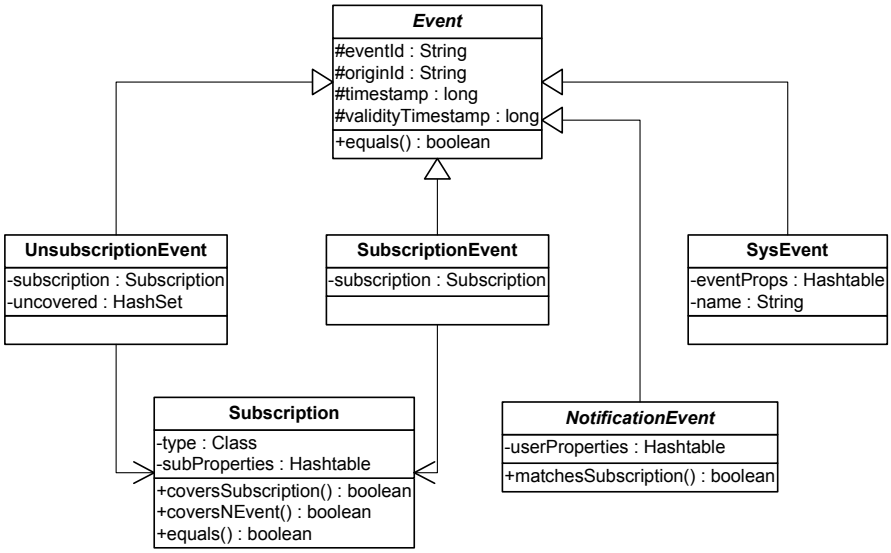

Figure 5.16: Class diagram of event classes

used only when starting the first broker: All added brokers and connected clients will receive the list of recognized types when connecting to an active broker. The type restriction represents no limitation with respect to other publish/subscribe systems. For example, JMS topics are administered objects that need to be defined by an administrator on the JMS server prior to being used by JMS clients.

Basic Classes

The vital system class is the abstract class `Event` that models events of the proposed publish/subscribe model. An `Event` object is uniquely identified by the field `eventId`; its creator is specified by `originId`; it carries the timestamp declaring the time of its creation, and the validity timestamp that announces when the event expires. Classes `NotificationEvent`, `SubscriptionEvent`, `UnsubscriptionEvent`, and `SysEvent` extend the class `Event` as depicted in the UML class diagram in Figure 5.16. The class `NotificationEvent` models the events *publish*, and *notify* from the publish/subscribe model, and carries the information published by publishers, either between a publisher and a broker, two brokers, or a broker and a subscriber. A `NotificationEvent` object can contain a list of user-defined attributes that are stored in the field `userProperties`. Note that each `NotificationEvent` object has a `validityTimestamp` field that determines its persistence in the system. All notifications published in the system need to extent the class `NotificationEvent`, either directly, or indirectly through parent classes.

Classes `SubscriptionEvent` and `UnsubscriptionEvent` model events *subscribe*, and *unsubscribe*, respectively. They contain a field `subscription` that defines the characteristics of user's subscription modeled by the class `Subscription`. A `Subscription` object determines the type of subscribed notifications as specified in the field `type`, and carries an optional

```
public boolean coversSubscription( Subscription sub ) {
  if ( this.type.isAssignableFrom(sub.getType()) ) {
      //check attributes
      ...
      return true;
  }
  return false;
}

public boolean coversNEvent( NotificationEvent nEvent ) {
  if (this.type.isInstance(nEvent)) {
      //check attributes
      ...
      return true;
  }
  return false;
}
```

Figure 5.17: The implementation of methods for checking the coverage relationship

list of attributes (`subProperties`) that further refine the subscription. The matching relationship between a notification and a subscription (defined in eq. 4.6) is implemented in the method `matchesSubscription()` of the class `EventNotification`, and `coversNEvent()` of the class `Subscription`. The covering relationship between subscriptions as defined in eq. 5.1 is implemented in the method `coversSubscription()` of the class `Subscription`. Figure 5.17 shows code fragments implementing the methods of the class `Subscription` that test the covering relationship. We use the methods provided by `java.lang.Class`: `isAssignableFrom()` checks the inheritance relationship between the two classes, and `isInstance()` tests whether an object is an instance of a given class. After checking the type, the methods compare the attributes, both attribute names and values, and in case all attributes of the covering subscription are present in the covered subscription, or notification event, the method returns `true`. The `validityTimestamp` for `SubscriptionEvent` and `UnsubscriptionEvent` is set to 0 by default: However, an option is given to initiate unsubscription to an active subscription after its validity period expires.

The class `SysEvent` models events *connect*, and *disconnect* of the publish/subscribe model, and enables a client to connect to, and disconnect from a broker. Connection to a broker does not mean a constant TCP connection between the two entities to preserve network resources. A new connection is initiated when it is needed. We use TCP instead of UDP to ensure reliable data transport. Therefore, each `SysEvent` object declaring a connection between a client and a broker carries a `validityTimestamp`, and maintains the client in the set of broker's connected clients until the timestamp is valid.

The infrastructure classes are `Client` and `Broker` that extend the class `Entity` as depicted in the UML class diagram in Figure 5.18. The class `Entity` is identified by its unique `id`. It uses an instance of the class `IncomingThread` that implements a TCP server socket listening on a defined port and accepting incoming messages from other `Entity` objects. The flag `inSystem` declares whether the entity is connected to the broker network or not, and `type` contains type definitions that

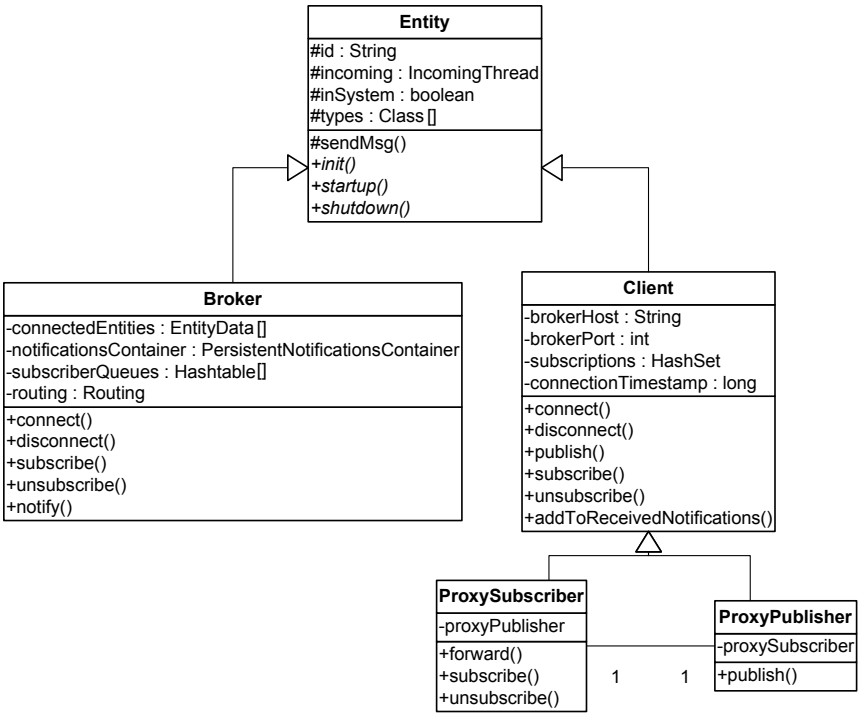

Figure 5.18: Class diagram of infrastructure classes

are used in the system.

The class `Client` models both publishers and subscribers: It contains the information about the broker to which the client is connected. The field `connectionTimestamp` defines whether the connection is still valid or not. The set of active subscriptions is stored in the field `subscriptions`, while the list of received and valid notification identifiers is recorded in the field `notifications-Container`. The class `Client` defines methods `connect()` and `disconnect()` that enable a client to send a message to a broker requiring connection or disconnection. The method `publish()` is used for publishing a notification event that is an instance of one of the defined notification types. Methods `subscribe()` and `unsubscribe()` enable a client to specify and send a subscription event or an unsubscription event to the broker. Special clients are `ProxySubscriber` and `ProxyPublisher` that implement the corresponding entities from the distributed publish/subscribe model.

The class `Broker` maintains a list of connected clients in the field `connectedEntities`. Connected clients can be either publishers, subscribers, or proxy publishers, and proxy subscribers that enable the communication between brokers. The information about the connected entities is described by a special class `EntityData`. Persistent notifications are stored in a special container `notificationsContainer`, queued notifications can be stored in a list of queues `subscriber-Queues`. Note that persistent notifications are used only if the PN-alg is applied. Queues are used for the Q-alg. Methods of the class `Broker` enable brokers to connect to, and to disconnect from other

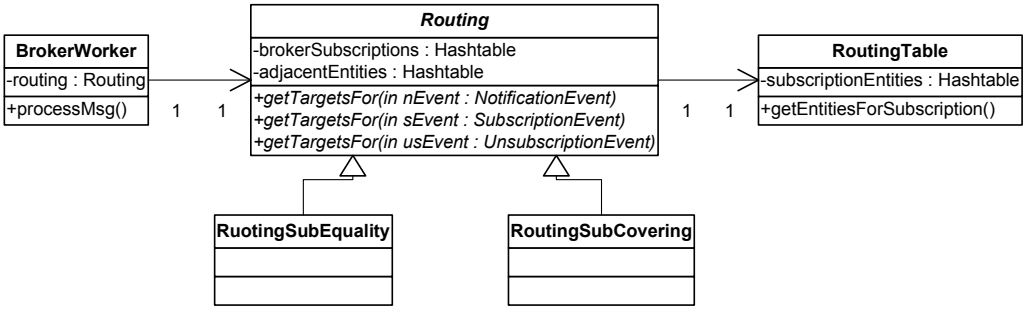

Figure 5.19: Class diagram of routing classes

brokers, to send subscriptions and unsubscriptions to their proxy subscribers, and to send notification to clients with matching subscriptions.

Implementation of the Routing Algorithms

The routing decisions regarding incoming messages received by brokers are made using the classes depicted in Figure 5.19. Each incoming message is processed in a new thread implemented by the class `BrokerWorker`. It implements methods for processing messages carrying the defined events. It uses the class `Routing`, i.e., the classes `RoutingSubEquality` and `RoutingSubCovering` that implement the corresponding routing algorithms, to make decisions about the neighboring entities to which the incoming message must be forwarded. Note that the system can be extended by new routing algorithms implemented by classes that extent the base routing class, and that the class `Routing` is a singleton [53], i.e., there can be at most one object instantiating the class in a running process. The class `Routing` maintains a list of active broker subscriptions in the field `brokerSubscriptions`, and a list of neighboring entities in `adjacentEntities`. `Routing` is associated with the class `RoutingTable` that maintains a mapping of each active subscription to a list of neighboring entities, either brokers or subscribers, that are subscribed to the particular subscription. `RoutingTable` is updated by each new subscription or unsubscription event, and speeds up the process of finding neighboring entities with matching subscriptions for incoming notifications. The usage of a routing table reduces the processing time needed to find subscriptions matching a published notification by avoiding program loops (for all subscribers, and for all their active subscriptions) and maintains a list of entities for each defined subscription. The routing table is updated with each subscription and unsubscription event.

The difference between the Q-alg and the PN-alg is not in the implementation of the class `Routing` and its subclasses, but in the procedures that are performed when a subscriber connects to a broker. In case of the Q-alg, the subscriber first activates its subscriptions to update the delivery path in the broker network, and later on retrieves the notifications stored in its queue maintained by the old broker. In case of the PN-alg, the subscriber also activates the subscriptions, and the broker network answers by routing persistent valid notifications matching the subscriptions to the subscriber. Prior

to notification delivery, the edge broker checks whether the subscriber has already received some notifications using the list of received and valid subscriber notifications. The next difference is in the notification storage: In case of the Q-alg, brokers store queues per each disconnected subscriber, while for the PN-alg, brokers maintain persistent notifications, and subscribers maintain a list of received and valid notifications.

The described set of classes builds the core of the publish/subscribe infrastructure. The application using the infrastructure needs a mechanism to use the infrastructure in a simple and transparent way. This is achieved through special interfaces: `ApplicationInterface`, and its extending interfaces `BrokerApplicationInterface` and `ClientApplicationInterface`. The interfaces define the methods that need to be implemented by application programmers using the MOPS infrastructure. The methods are invoked if an entity, either a broker or a client, receives an event. They notify the application layer about event occurrences. The detailed description of the application interfaces and an example application is presented in [75]. The detailed description of the system implementation based on the PN-alg is given in [117].

5.3.2 Queuing Algorithm vs. Persistent Notification Algorithm

This section presents experimental results that assess the performance of the PN-alg, and compares the PN-alg to the Q-alg. The main differences between the Q-alg and the PN-alg are in the following:

- **Notification storage.** In case of the Q-alg, system brokers store queues per each disconnected subscriber. For the PN-alg, brokers maintain persistent notifications, and the list of valid notifications sent to subscribers and neighboring brokers.

- **Subscriber's reconnection to the system.** When applying the Q-alg, a reconnecting subscriber first reactivates its subscriptions at the new broker to update the delivery path in the broker network, and next, the new broker retrieves the notifications from the subscriber's queue maintained by the old broker and delivers them to the subscriber. In case of the PN-alg, the subscription reactivation will initiate the delivery of valid notifications along the new delivery path to the subscriber. Prior to notification delivery to the client, the edge broker checks whether the subscriber has already received a valid notification by comparing it to the list of received notification ids.

- **Subscriber's data.** A subscriber in the system applying the Q-alg needs to know the identifier of the old broker together with the list of active subscriptions to reconnect to the system. In case of the PN-alg, a list of received and valid notification ids, and the list of active subscriptions is needed.

- **Perceived number of system subscribers.** Subscriber queues act as proxy subscribers for disconnected clients which gives the impression that subscribers are constantly active in a system that uses the Q-alg. The PN-alg maintains no active subscriptions for disconnected subscribers.

The evaluation results are obtained using a working prototype which emulated the real working environment, instead of model simulation. There are some differences in the implementation of the two approaches that cause the increased processing load for system brokers and clients in case of the PN-alg: The main reason is the implementation of a garbage collector that purges expired notifications from notification containers. Therefore, we have decided to define the metrics that are not largely influenced by the processing latency to enable a just comparison of the two approaches.

Metrics. The performance of a publish/subscribe system in a dynamic environment with mobile clients is largely influenced by its efficiency: minimal processing load on the brokers, minimal bandwidth consumption, and minimal notification delay. We propose the usage of the following metrics for mobile publish/subscribe system evaluation:

- **Broker processing load.** The processing load experienced by a broker can be measured by the rate of processed messages. Messages carrying notifications transport the actual information, while subscription and unsubscription messages represent control load that creates and updates delivery paths. We differentiate between received and sent messages, and classify them according to the type of events they transport.

- **Bandwidth consumption.** A desirable property of a distributed pub/sub system is to consume minimal bandwidth. The rate of processed messages, as in the case of broker processing load, gives a good estimate of the physical bandwidth consumption.

- **Notification delay.** Efficient notification delivery requires minimal delay, i.e., the period between notification publication and receipt. In case of mobile subscribers, the delay is increased due to subscriber disconnections from the system, and it depends on the duration of disconnection periods and notification validity periods.

Best to the author's knowledge, this is the first evaluation of publish/subscribe system performance in a mobile setting that provides performance measures regarding the broker load, notification delay, and bandwidth consumption. The results presented in [21] evaluate the mobility implementation within the project Siena. The results show that the extended mobility-enabled system functions correctly, i.e., that notifications published during the disconnected period reach subscribers as they reconnect to the system. The authors investigate the number of duplicate and lost messages, but define no other metrics to evaluate system performance.

Experimental Setup

The experiment investigates the broker's processing load, and bandwidth consumption of the Q-alg and the PN-alg in terms of the rate of processed messages, and the number of stored notifications. Furthermore, we investigate the implementation efficiency in terms of delay. We ran the experiment under the same initial conditions for the Q-alg and the PN-alg. After forming a network of brokers, we initiated stationary publishers, and the defined set of mobile subscribers. Each experiment run lasted 15 minutes, and we conducted 5 runs with the same initial setting. Therefore, each data point in

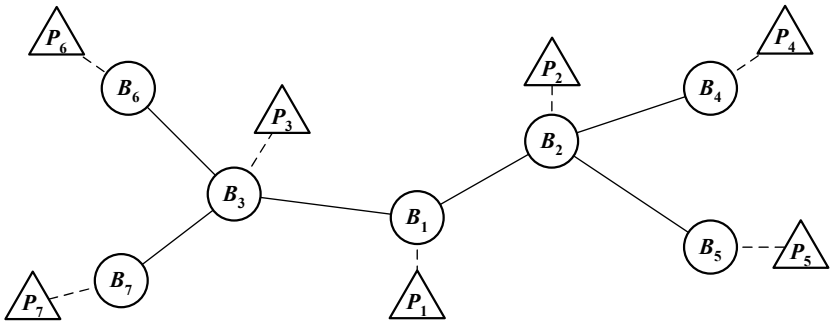

Figure 5.20: Experimental network

the given charts is an arithmetic mean of 5 runs. The experiment was conducted using two computers (Celeron 2.2 GHz, 512 MB of RAM) running Windows XP. The first computer was used to set up the network of brokers, while the second one hosted both publisher and subscriber processes.

Input parameters. The experiment is conducted using a stationary network of seven brokers forming a tree as depicted in Figure 5.20. The number of publishers is constant $p = 7$, and each publisher is stationary and connected to one of the brokers. Each publisher publishes notifications at a constant rate of $pubRate = 0.5$ notifications/s. We use a complex type hierarchy consisting of 20 types, and publishers generate notifications of a randomly chosen type with uniform probability. Each notification carries a payload of 100 bytes. The validity period for notifications in case of the PN-alg is set to 5000 ms to avoid potential undelivered notifications.

We varied the number of subscribers in the system, $s = 1, 5, \ldots, 30$. Subscribers are mobile and can connect to all system brokers except to B_1 because it is the root node of the broker network, and therefore the system bottleneck. We use the random mobility model [66] in the experiment: A subscriber chooses the next broker randomly from the set of available brokers. Each subscriber connects to a new broker with a constant connection rate in the range from 0.2 to 0.6 connections/s. Connection duration is 50% of the connection period. For example, when $connRate = 0.2\ 1/s$ and $connPeriod = 0.5$, a subscriber is connected to a broker for 2.5 s, then it disconnects from the system, and after 2.5 s reconnects to a new broker. Figure 5.21 shows the measured average number of subscribers connected to a broker as the total number of subscribers in the system changes. It is visible that subscribers do not connect to B_1, and that other brokers evenly share the subscriber load. In case of the total of 15 subscribers in the system, there is on average one subscriber connected to each broker. All subscribers subscribe to the top subscription type, and should receive all published notifications. The overview of input parameters for the experiment is given in Table 5.1.

Experimental Results

The rate of received and sent messages. Figure 5.22 shows the average rate of received and sent notification messages per each broker. The rate of *received notification messages* for both the Q-alg

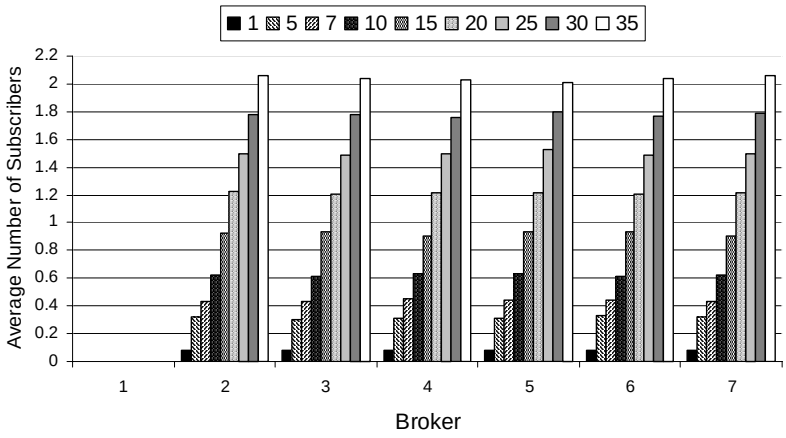

Figure 5.21: Number of connected subscribers per broker

Table 5.1: Input parameters

Publishers		**Subscribers**	
p	7	s	1, 5, 10, 15, 20, 25, 30
$pubRate$	0.5 1/s	$connRate$	0.2 – 0.6 1/s
$loadSize$	100 bytes	$connPeriod$	0.5
$validity$	5000 ms	brokers	$B_2, B_3, B_4, B_5, B_6, B_7$

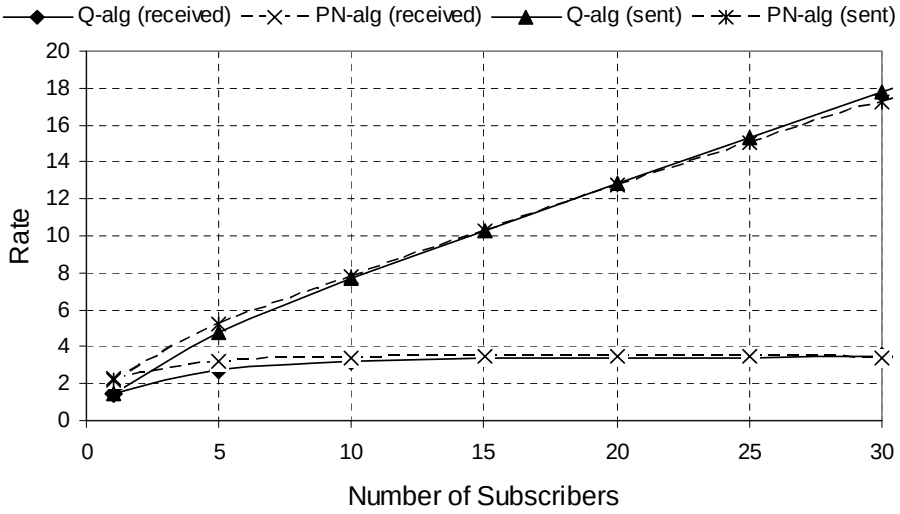

Figure 5.22: Rate of received and sent notifications

and the PN-alg increases until $s = 15$ when there is on average 1 subscriber per each broker, and reaches the maximum value determined by the number of system publishers, and their publishing rates, e.g., for $p = 7$ and $pubRate = 0.5$, the maximum rate of received notifications for each broker equals 3.5 when each broker receives all published notifications. Clearly, the rate of *sent notification messages* increases as the number of subscribers in the system increases, because the number of message destinations increases accordingly. The rate of sent notifications is not significantly different for both approaches in the experiment. However, the Q-alg generates a slightly larger number of notification messages than the PN-alg when $s \geq 15$, because in case of the Q-alg, notifications are sent to subscriber queues during the disconnection period, while in case of the PN-alg, notifications are cached on brokers they traverse, and from there delivered to reconnecting subscribers. The notification rate in case of the Q-alg is further increased by notification exchange during the handover procedure. The difference is not significant for 30 subscribers in the system, but the trend shows that it would be substantial in case of a large number of subscribers.

Figure 5.23 shows the average rate of received and sent subscription messages per each broker. As expected, the rate of *received subscription messages* increases for both algorithms as the number of subscribers in the system increases, and the rate of *sent subscription messages* decreases due to the existence of covered subscriptions in the system as the number of subscribers increases. The graph shows that there are no significant differences between the two approaches as the number of subscribers in the system increases.

Figure 5.24 shows the average rate of *received and sent unsubscription messages* per broker. As expected, the Q-alg generates less unsubscription messages than the PN-alg because the Q-alg does not generate unsubscription messages in case the old and the new broker are the same, which is the case for the PN-alg. The rate of sent unsubscription messages decreases as the number of subscribers

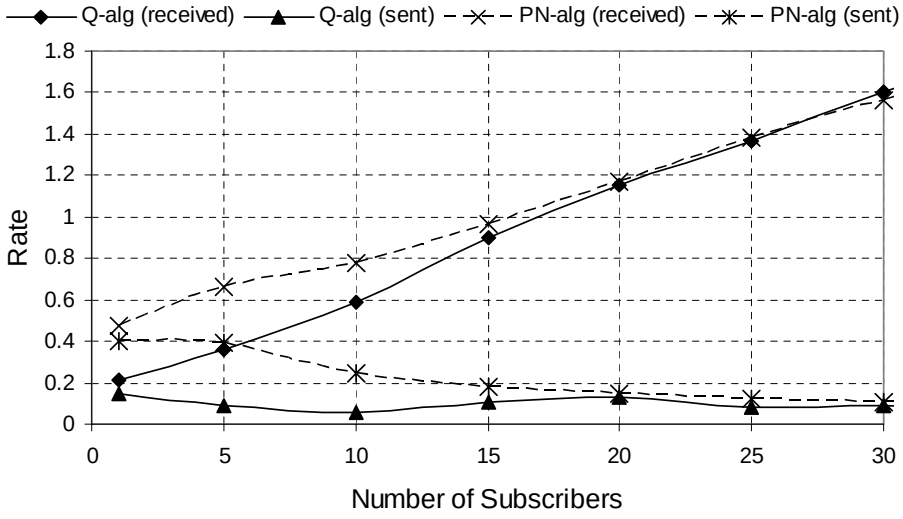

Figure 5.23: Rate of received and sent subscriptions

increases, because there is no need to propagate unsubscriptions since there are other subscribers with an active matching subscription that are connected to brokers.

Figure 5.25 shows the average rate of *control messages* per each broker that create and update delivery paths in the broker network. We refer to both subscription and unsubscription messages as control messages. Although it would be reasonable to assume that the rate of control messages in case of the PN-alg is substantially larger than for the Q-alg due to the increased number of unsubscriptions, the experiment shows that as the number of subscribers in the system increases, the difference between the PN-alg and the Q-alg decreases.

Finally, Figure 5.26 depicts the average rate of all received and sent messages per broker and can be used to asses the broker processing load. The messages that are taken into account are notifications, subscriptions, and unsubscriptions. It is visible that the Q-alg poses less load on a broker for a smaller number of subscribers, but that the performance of the PN-alg improves as the number of subscribers in the system increases.

The number of stored notifications. Figure 5.27 shows the average number of routing table entries on a broker as the number of subscribers in the system increases. The size of routing tables are important as they directly influence the routing efficiency and delivery delay for published notifications. As expected, the number of routing table entries is larger for the Q-alg than for the PN-alg because the Q-alg experiences a larger number of subscribers in the system because disconnected subscribers are represented by queues and require routing table entries. It can be concluded that the PN-alg is superior when compared to the Q-alg with respect to the number of routing table entries especially if we assume that the number of subscribers in the system is large, and if disconnection periods are frequent and long.

Notification delay. Figure 5.28, Figure 5.29, and Figure 5.30 show the experienced delay per

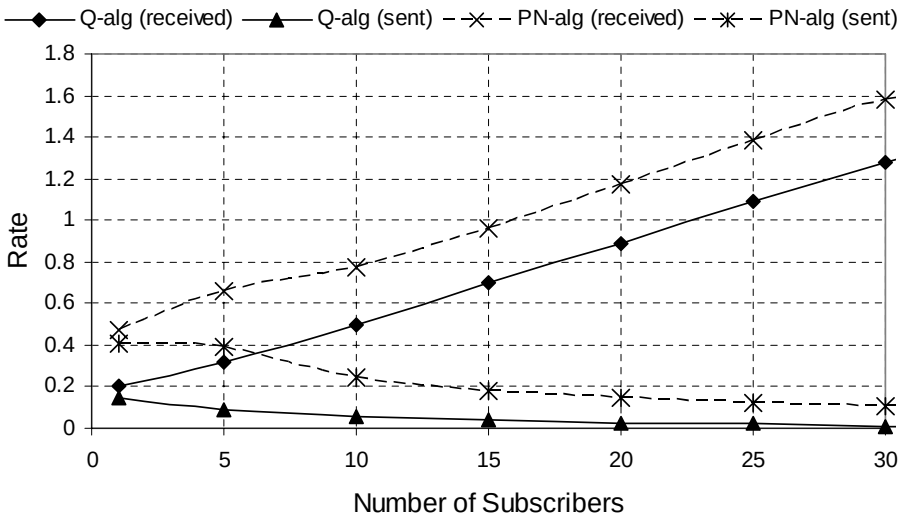

Figure 5.24: Rate of received and sent unsubscriptions

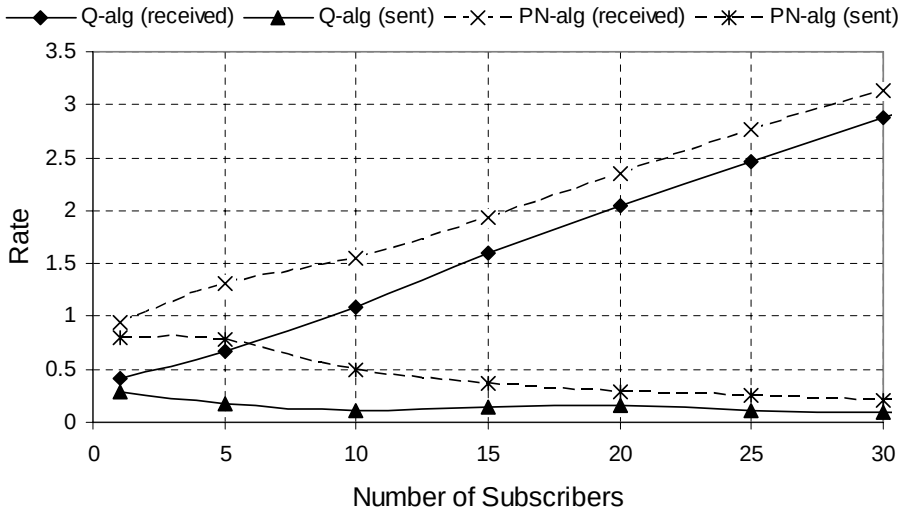

Figure 5.25: Rate of received and sent control messages

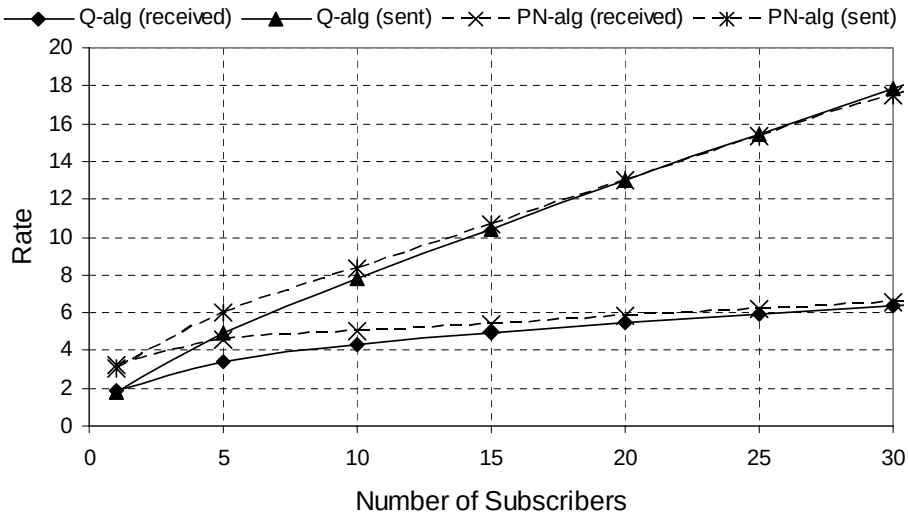

Figure 5.26: Rate of received and sent notifications/subscriptions/unsubscriptions

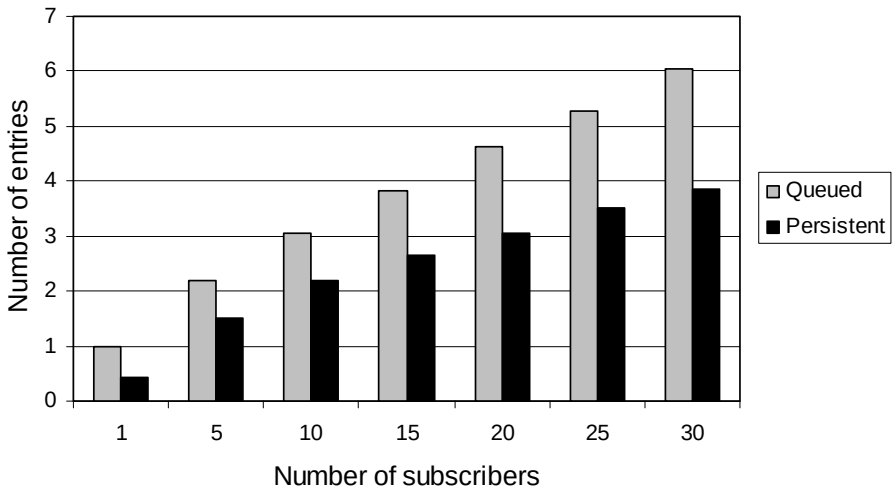

Figure 5.27: The average routing table size per broker

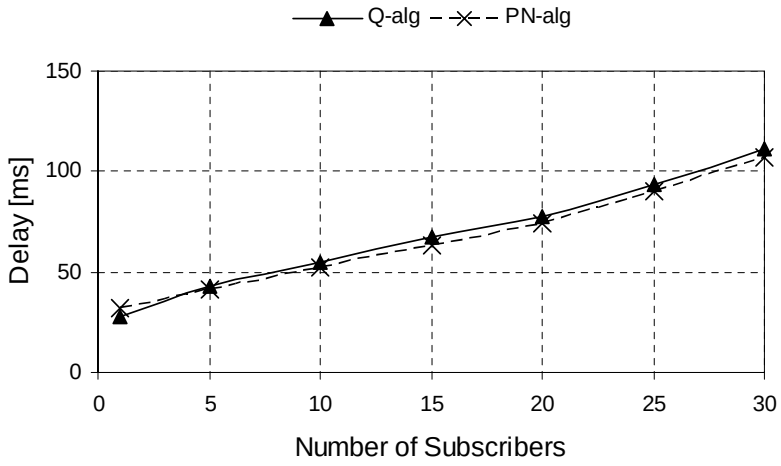

Figure 5.28: Delay for direct notifications

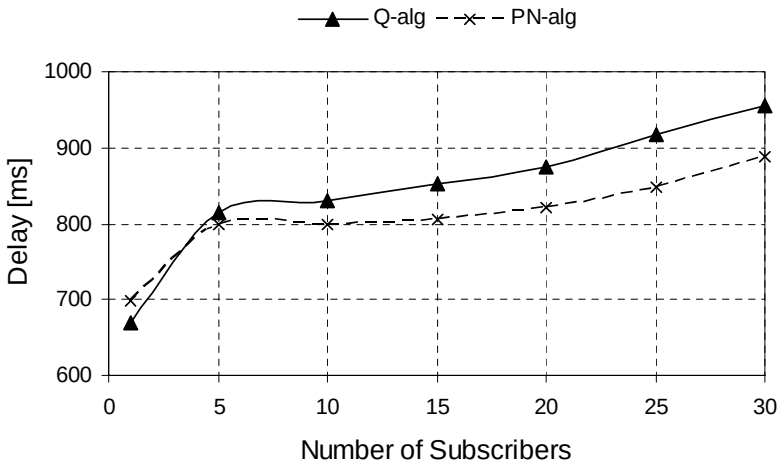

Figure 5.29: Delay for stored (queued/persistent) notifications

received notification in ms as s varies. Notifications taken into account in Figure 5.28 were directly delivered to subscribers without being stored in the system. The delay of stored notifications, either queued, or persistent, is depicted in Figure 5.29. Figure 5.30 shows the average delay for all published and delivered notifications. The figures show that the PN-alg causes smaller delay than the Q-alg as a consequence of smaller routing tables. The delay is also influenced by the differences in broker implementations: PN-alg brokers are burdened by special processing threads for updating the list of unexpired notifications, and therefore the difference between the two approaches is less significant.

To conclude, the PN-alg is superior when compared to the Q-alg with respect to the routing efficiency as it generates smaller routing tables and introduces smaller notification delay. The PN-alg introduces less processing load on brokers for the conducted experiment as the number of subscribers in the system increases, and therefore consumes less bandwidth on links connecting the brokers. The

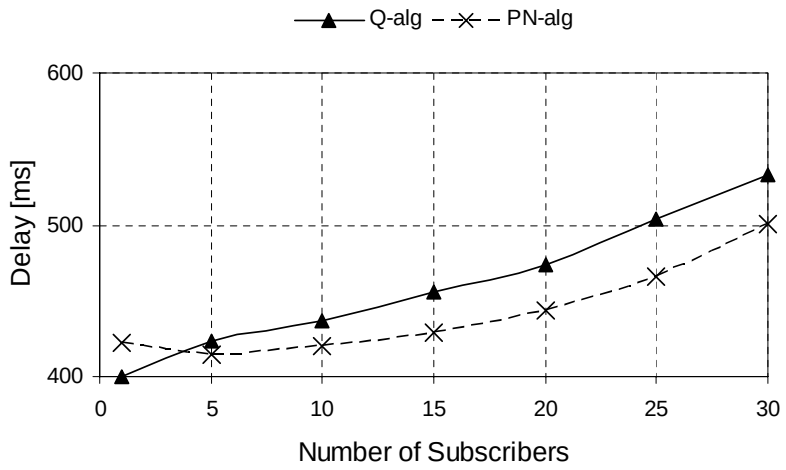

Figure 5.30: Delay for all notifications

preliminary results show that the load introduced by control messages for the PN-alg is acceptable when compared to the Q-alg.

5.4 Discussion

Mobility and persistent notifications. The usage of persistent notifications stored by the system, and the algorithm that requires the delivery of such notifications after subscriber's reconnection to the system, is a novel approach that assures the delivery of notifications published during subscriber's disconnections. When a subscriber reconnects to the system, possibly through a new broker, the new broker reactivates its subscriptions. The subscriber needs to provide a list of its active subscriptions, and a list of subscriber's valid received notifications. The broker can alternatively retrieve the list of subscriptions and received notifications from the broker network. The list of active subscriptions is needed to reactivate subscriptions, and the list of subscriber's valid received notifications prevents delivery of duplicate notifications. The reactivation of subscriptions causes the creation of a new delivery tree to the subscriber using the new broker as the root node of the delivery tree. The rule that requires the delivery of persistent valid notifications after each subscription reactivation assures that the subscriber receives persistent notifications stored by the broker network that it has not previously received.

A persistent notification is stored by a subset of network brokers until its validity period expires. It is maintained by a single broker if, at the time of its publishing, there were no remote subscribers for the notification. The notification will eventually reach a reconnecting subscriber following a newly-created delivery path, and be stored on each broker it traverses. The subscriber might have already received the notification if it has previously resided on the broker through which the notification has been published: The notification will be routed to the new broker, however, it will not be delivered to the subscriber since its id is in the list of subscriber's received notifications. This in the known

overhead of the approach that causes superfluous traffic in the broker network, and increases the usage of broker memory and processing time. At the other extreme is the situation in which all brokers have a notification copy. A reconnecting subscriber will receive a notification copy from the access broker without causing extra traffic in the broker network.

If we assume reliable communication in the system, the system is resilient to duplicate notifications. The mechanism that compares the list of subscriber's received notifications that are still valid to those that should be sent to a subscriber prevents the possibility of delivering duplicate notifications. However, a notification can be undelivered because a broker network may introduce the delay that can lead to notification expiry prior to it's delivery to the subscriber.

Comparison with the "queuing" approach. Potential advantages of the proposed approach when compared with the Q-alg are the following: avoidance of the handover procedure that transfers notifications from the old to the new broker, reduced size of broker routing tables due to decreased number of perceived subscribers in the system, and memory consumption related to the storage of notifications in the system. The expected disadvantage is related to control traffic: The PN-alg generates an increased number of subscriptions and unsubscriptions for terminating the old and creating the new delivery paths. The Q-alg suffers from the same problem if the probability that a subscriber reconnects to the same broker is low. If subscribers frequently connect to the same broker, the number of control messages is reduced because there is no need to update an existing delivery path. The maintenance of the list of received and valid notification ids is an additional broker overhead in case of the PN-alg.

Advertisements. In the presented algorithms we utilize only subscription and notification messages: Subscriptions create delivery paths for notifications. Some systems, for example SIENA [24] and Hermes [89] use advertisements to decrease the number of control messages exchanged between system brokers. Advertisements are published by notification publishers to declare the intent of publishing notifications with certain characteristics. Advertisements set routing paths for subscriptions: Every advertisement is broadcasted to all brokers, and subscriptions follow reverse paths set by advertisements to set delivery paths from potential publishers to subscribers. A subscription is routed to a neighboring broker only if it advertises notifications matching a subscription. The forwarding of advertisements decreases the number of control messages that update subscription information since subscription update is sent only to those brokers that can generate matching notifications. The algorithms presented in the thesis can be extended to use advertisements for decreasing the rate of subscription and unsubscription messages in the system.

Reverse path forwarding vs. core-based trees. The algorithms use reverse path forwarding for creating minimal delivery trees for notifications and offer minimal dissemination delay, but increase the number of control traffic in the broker network. It is possible to adjust the presented algorithms to rely on the core-based tree approach for routing notifications to mobile subscribers. Subscribe and unsubscribe requests would be routed to the core node creating and updating delivery paths from the core node to subscribers. Each published notification would be routed to its core node, and from there it would follow delivery paths to subscribers. The notification would be stored at least by the core node until it is valid.

A comparison of the routing algorithm using the core-based tree approach that is applied in Hermes and the reverse path forwarding algorithm similar to the one used in Siena is given in [90]. The comparison considers environments with stationary clients. The algorithms are compared with respect to the following cost metrics: delay, bandwidth consumption, and routing table sizes. The experiments show that the reverse path forwarding approach using minimal spanning tree is superior to the core-based approach with respect to notification delay, i.e., the average time between notification publication and delivery. The core-based tree approach uses less space for storing routing tables because the routing information is not flooded through the network, but rather forwarded to core nodes. The bandwidth consumption is measured in terms of the number of messages exchanged between network brokers: The core-based tree approach generates less notification and advertisement messages, but creates more subscription-related traffic because of additional messages sent to the core node. The performance of the core-based tree routing approach needs further performance investigation, especially in mobile settings.

An outline of a distributed JMS implementation. The current JMS-based distributed implementations, e.g., JORAM [84], use the flooding technique for disseminating notifications between JMS servers. This is a costly technique that causes significant bandwidth consumption and processing load, and may interfere with system scalability. The proposed distributed publish/subscribe model and the routing algorithms with persistent notifications are adequate for transforming centralized JMS solutions into scalable distributed JMS implementations: Firstly, the extension of an existing server can be implemented by adding a pair of proxy publishers and proxy subscriber on top of two JMS servers in order to enable equality-based or covering-based routing between the two JMS servers. Secondly, the JMS specification defines message persistency, and it has been integrated into the available JMS implementations. Thirdly, JMS uses open connections for its publishers and subscribers: It is therefore possible to update the set of connected subscribers which is needed for the algorithm implementation.

The available JMS benchmark tests [50] show that JMS implementations are largely influenced by the number of connected subscribers, and the publishing rate. The extension of the existing server implementation would improve the performance of the distributed solution, and enable the deployment of scalable distributed solutions that can satisfy requirements for large-scale notification systems. Further work is needed to implement and evaluate the performance of distributed JMS implementations based on the presented model and routing algorithms.

Chapter 6

Content Dissemination Service Architecture

The increasing popularity of information services that rely on content delivery in mobile environments motivates the need for a *mobile content dissemination service*—an efficient and scalable information service that enables the delivery of personalized and customized content to mobile users. Publish/subscribe middleware offers mechanisms for content personalization: Subscribers define the characteristics of content that is of interest to them, and get notified when such content becomes available. The distributed architecture of publish/subscribe systems and efficient routing algorithms solve the scalability requirement. However, the diversity of usage scenarios and the varying nature of mobile environments requires additional services that need to cooperate with the publish/subscribe middleware to offer a flexible service customized to particular user presence status.

This section lists and analyzes the requirements of a content dissemination service supporting mobile users, and present representative usage scenarios that illustrate the features of the system offering customized content dissemination to mobile users. The analysis of service usage scenarios has enabled the identification of the supporting software components that need to collaborate with the mobility-aware publish/subscribe middleware to address the dynamics of mobile environments. The set of identified components forms the proposed content dissemination service architecture.

The chapter is structured as follows: Section 6.1 investigates usage scenarios that have guided the design of the reference architecture which we present in Section 6.2. The reference architecture has a layered structure comprising a set of components: We outline the features of the identified components and describe their interaction in Section 6.2.4. We focus on two particular components that are crucial for service implementation and deployment: publish/subscribe middleware, and personal mobility management. We provide a detailed design description of a Web-based publish/subscribe component in Section 6.3. Section 6.4 illustrates our solution for personal mobility management.

6.1 Requirements and Usage Scenarios

Even though content dissemination services can be useful, their wide acceptance depends on the precondition that the service delivers only highly personalized and customized content in accordance with user preferences and current presence status. This gives the opportunity to create a "branded" dissemination service invulnerable to spam. The service would become a trusted intermediary between content publishers and subscribers that filters the wealth of information according to user's needs. We identify the following requirements that need to be satisfied by service design:

Push-based content delivery. Service users must be able to define the type of content they want to receive, and be served with the published information as soon as it is available. The push-style content delivery eliminates the burden of querying for information at regular intervals and is in accordance with the stochastic nature of content creation and publication.

Content filtering and personalization. Content filtering is enabled through user subscriptions to minimize the number of received message that are not of interest. This feature enables service personalization and adaptation to user context. It reduce the information overload on a user by associating and comparing each published piece of information to user context and preferences.

Personal mobility. Service users must be able to publish and receive the content using various terminals in different networks. This feature enables true personal mobility and offers usage flexibility.

Scalability. The service must scale well to a large number of potential users, and must be optimized for the particular application area with respect to the number of publishers and subscribers in the system, and the size and frequency of published content.

We describe a number of usage scenarios for content dissemination services in mobile environments: We start the analysis with the simplest scenario that offers no mobility support and gradually extend it to introduce more flexibility for service users. In the first scenario a user employs a stationary terminal with a permanent network attachment point to publish and receive the content. The second scenario enables nomadic users to access the service from different networks using desktop or portable computers via dial-up modem lines or (wireless) LANs. In the third scenario a user can apply various devices ranging from desktop and laptop computers to less powerful devices such as handheld computers and mobile phones in different networks. We consider the content dissemination system as a black box and put the user, either a content publisher, or a subscriber, in the focus of our discussion.

We use the following underlying scenario to motivate the discussion and illustrate the usage scenarios: Alice lives in the suburbs of Zagreb and commutes each day to her downtown office. She uses the *traffic notification service* which informs her about the current traffic situation. The up-to-date traffic reports enable Alice to decide whether to rely on public transportation or to drive to work. The service can assist her in finding the best driving route.

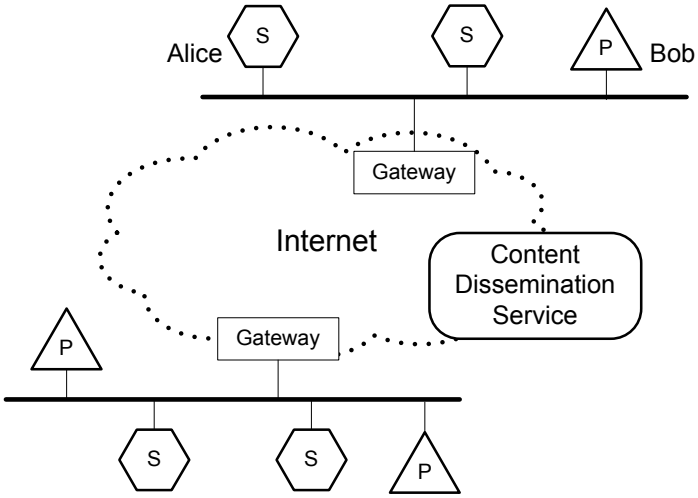

Figure 6.1: Stationary scenario

Stationary users. Alice accesses the traffic notification service from her office desktop computer on a LAN. Before leaving the office, she checks the list of received traffic reports to be informed about the current traffic situation. If she needs additional information and driving instructions, she can request a detailed map of the particular area with approximate waiting times for the traffic jam areas.

Figure 6.1 depicts a stationary scenario showing an environment hosting the traffic dissemination service. The service is deployed in the IP-based network, while service publishers and subscribers apply stationary terminals for publishing and receiving the traffic reports. In this scenario Alice is a stationary subscriber to the channel "Zagreb traffic." Whenever traffic problems in the area of Zagreb are reported, for example by Bob who has just arrived to the office, the traffic service initiates the delivery of a new traffic report to all subscribers with a matching subscription. This is a standard push-style service operation, while the use case in which Alice requests additional information using, for example, an URL from the received report, relies on the request/reply interaction style.

In the stationary scenario all subscribers apply a single static terminal for receiving the published content: If a terminal has a *public stationary IP address*, the broker hosting the traffic dissemination service can initiate connections to the content receiver running on the terminal to deliver a published report. For this purpose the broker must know the address of the terminal and whether the receiver application is running on the terminal. In case the terminal is behind a firewall, i.e., it has a *private IP address*, the broker cannot initiate a connection to the receiver: The receiver must initiate a connection to the broker and maintain it active. The broker can deliver the published content using the active connection to the terminal behind a firewall.

In case a traffic report cannot be delivered to a subscriber's receiver application, the undelivered report must be stored by the service for subsequent delivery. The service needs to provide a strategy for *temporary content storage* that will preserve undelivered content for disconnected users according to

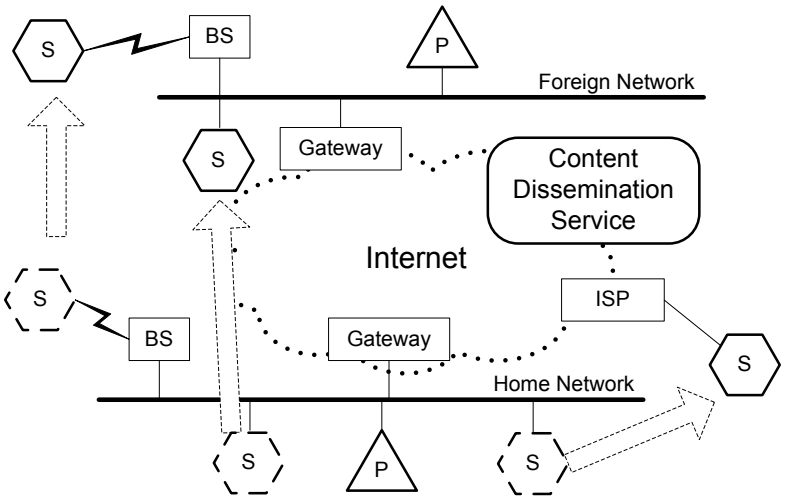

Figure 6.2: Nomadic scenario

the defined priority rules. Furthermore, the service enables publishers to define the topics for content classification, and to define and store the content for publishing using a *content management service*.

We have identified personalization as one of the major requirements of a dissemination service. For example, Alice might define several routes between her home and office. In this case the traffic service would filter the messages for the "Zagreb traffic channel" and deliver only those that match her personal routes. Clearly, *content-based filtering* is needed to provide such a personalized service and Alice must also be able to express her preferences as a set of rules/filters. Her subscriptions and preferences are stored and managed by a *user profile service*.

Nomadic users. In the previous scenario Alice was restricted regarding the usage of the traffic notification service since she was bound to one location. Naturally, she wants to use this service at home before driving to the office. At home she connects her laptop to the Internet via dial-up and thus becomes a nomadic service user.

This feature puts an additional requirement on the service: The service must be aware of the end communication point, e.g., a terminal address and a port number, to deliver the content to a subscriber. In case a nomadic user applies a single terminal with support for mobile IP, the mapping remains stationary as in the case of stationary users. However, if the user applies a terminal without mobile IP support, or various terminal, the service needs an up-to-date information that uniquely defines the end communication point because if the content is sent to an invalid address it might reach the incorrect subscriber or the service might assume that a subscriber is off-line. Further on, a subscriber may apply various applications for receiving the content, e.g. a JMS-based receiver, or a mail reader, and the service should be able to deliver it using the most appropriate and preferred delivery mode.

A nomadic user can frequently change its location in the network even though the service is not used between the movements. Figure 6.2 depicts a nomadic scenario in which the terminal changes the network, or a subscriber changes both the device and network. A subscriber can use the service

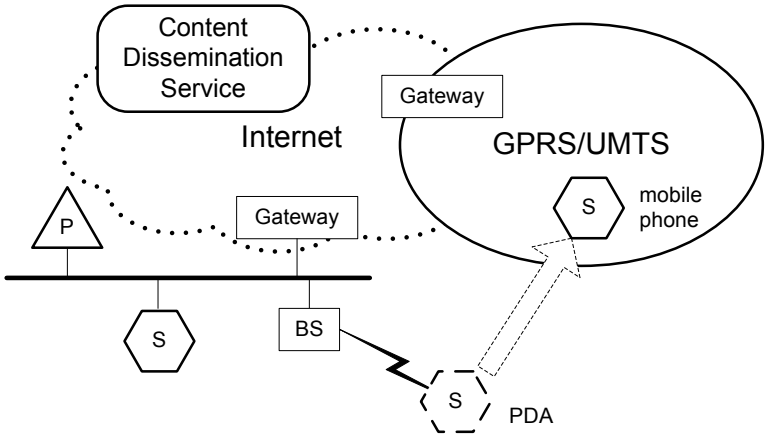

Figure 6.3: Mobile scenario

from a dynamically configured home network, or move to a foreign network and connect to the Internet via wireless LAN. A subscriber can also use the service from home via dial-up. By changing its attachment point, the terminal address and the subscriber's communication point will change accordingly. The same problem arises if a network (LAN, PPP) is configured using the Dynamic Host Configuration Protocol (DHCP). To track the change of the subscriber's current destination address, a *personal mobility management service* must map a unique subscriber identifier to the current subscriber's communication point. We assume that the communication point identifies both the terminal and the application for content receipt.

Mobile users. In this scenario Alice would like to use the traffic notification service while in motion. Figure 6.3 depicts an environment where she can use both a handheld computer and a mobile phone to receive traffic reports. She can use a handheld computer with wireless LAN connectivity while within the reach of a wireless LAN base station or her mobile phone during outdoor activities.

In this setup, as well as in the nomadic scenario, we need a *personal mobility management service* that will map a user to the identifier of the currently applied end communication point. This is a one-to-many mapping: A user might register a number of devices and receiver applications, e.g., a mobile phone with MMS, a handheld computer with a mail reader, a desktop, or a laptop computer with a JMS-based receiver. In case a user is applying a single terminal with multiple network interfaces and a single receiver application, the network should provide *vertical handover* [18], a smooth transition between different networks. For example, when a user exits the wireless LAN coverage, the terminal automatically switches to the mobile network.

Service personalization and *content-based filtering* are vital in this scenario because a user must be able to define his/her preferences according to the currently used communication point. For example, Alice may want to receive traffic reports regarding the area in which she is currently residing when using her mobile phone, while in case she is in the office, she would need all the published information about the Zagreb area.

Table 6.1: Services for stationary, nomadic and mobile users

	Stationary	**Nomadic**	**Mobile**
content-based filtering	+	+	+
temporary content storage	+	+	+
content management	+	+	+
user profiles	+	+	+
personal mobility management	-	+	+
content adaptation	-	-	+
content presentation	-	-	+

Due to the variations in network and end-device, *content adaptation and presentation* is essential in this scenario. The content is delivered through various networks that differ in the available bandwidth, and it is displayed on devices with different computational capabilities and screen sizes. For example, Alice can receive high quality maps only on a computer with a high bandwidth connection. When driving home from the office she can re-check the text reports about the changing traffic conditions on her mobile phone. The published content depends on the terminal and the network both publishers and subscribers are applying. Similarly, the presentation of the received content will depend on the characteristics of the subscriber's terminal. The content must therefore be adapted to match the capabilities of both terminal and network.

Table 6.1 summarizes the required services for each of the described usage scenarios. Content-based filtering is needed for service personalization in all usage scenarios. Temporary storage of undelivered content must be provided to guarantee the delivery of valid and possibly vital reports to disconnected users in all usage scenarios. Content publishers define topics for content classification and the content for publishing using the content management service, while subscriber's subscriptions and preferences are managed by the user profile service. Both content management and user profile service are needed in all scenarios. Personal mobility management is required in nomadic and mobile scenarios, while solutions for content adaptation and presentation become important in mobile scenarios with different devices and networks. It is desirable to design and implement generic services that can accommodate the requirements of different scenarios and therefore be applied in all presented scenarios.

6.2 Reference Architecture

Based on the discussion in Section 6.1, we propose a content dissemination service architecture for mobile environments that is based on the publish/subscribe communication infrastructure. Figure 6.4 depicts the proposed architecture that consists of the components providing the features listed in Ta-

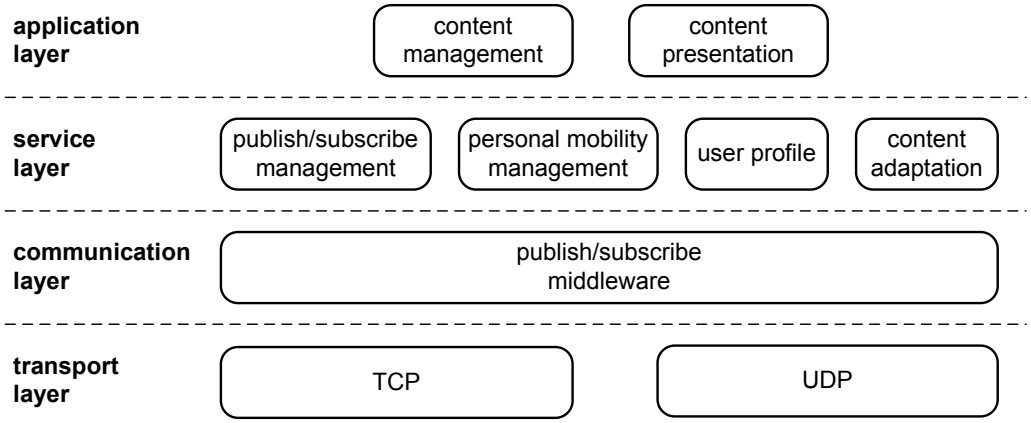

Figure 6.4: Reference architecture

ble 6.1. We denote the proposed architecture as *reference architecture* because it identifies a complete set of components needed to provide a personalized and adaptable content dissemination service for mobile environments. This section gives an overview of component characteristics and describes their functionality.

The components are logically divided into the following layers:

- The *communication layer* enables the publish/subscribe interaction between system users and other services that require event-based communication. It employs publish/subscribe middleware that provides push-based content delivery, content-based filtering according to defined subscriptions, and temporary content storage for disconnected subscribers.

- The *service layer* contains utility services needed by a content dissemination service. These are publish/subscribe management, personal mobility management, user profile service, and content adaptation.

- The *application layer* is a service-specific layer that deals with content presentation, and enables publishers to define and manage device-dependent content.

The presented architecture adopts the approach used in the Open Service Access (OSA) standard [1]. OSA specifies an open standard API for third party service providers that enables them to design and deploy value-added services using the network infrastructure controlled by mobile operators. OSA offers abstractions of the core network functionality through OSA services, e.g., user location service, user status service, call control, user interaction service, terminal capabilities service, presence and availability management. OSA services are deployed in a layer above the network infrastructure: They rely on the network communication and management services that have traditionally been unavailable to third party service providers. Security and authenticated access to network services is obviously the major requirement that needs to be fulfilled for OSA-based service deployment in real networks.

Following the OSA approach, we define generic services in the service layer of the reference architecture. The identified services use the publish/subscribe communication capabilities of the underlying layer: The publish/subscribe middleware may be deployed as a Web service and offered and managed by a third party. We propose the design of such a publish/subscribe service in Section 6.3. Some of the OSA services correspond to services in our reference architecture: Personal mobility management corresponds to the OSA presence and availability management service, and user status service; user profile can be mapped to the OSA interaction service. Assuming that the OSA implementation is available and deployed in the existing mobile networks, service providers can use OSA services to implement and offer content dissemination: OSA services would enable access to the infrastructure and data of the mobile network. However, OSA services are still not supported and offered in the existing networks. Furthermore, the service that is missing in the OSA standard is the publish/subscribe-style communication which is required for the presented content dissemination solution.

6.2.1 Communication Layer

Publish/subscribe is the basic interaction style in the proposed architecture. It enables the push-based delivery of content from publishers to subscribers, and event-based interaction between other architecture components. Subscribers can define subscriptions to channels and refine their subscriptions using the set of rules for content-based filtering. We use the term *channel* for content classification to avoid implying the subscription scheme of the applied publish/subscribe system: The underlying publish/subscribe middleware may implement either subject-based, content-based, or type-based subscription schemes.

To offer support for mobility, the publish/subscribe middleware should provide temporary storage of published content for disconnected subscribers using either the “queuing” approach, or the storage and delivery of persistent notifications. In addition, it serves as a distribution media for notifications produced by a mobility management component, and environment-related events that guide service adaptation. We assume that the publish/subscribe system has a distributed architecture to cope with scalability-related problems and propose that it is designed and implemented using the distributed mobility-enabled model presented in Chapter 4. The detailed description of the communication layer design is given in Section 6.3.

6.2.2 Service Layer

The *publish/subscribe management* component is a mediator between the application layer and the publish/subscribe middleware. It is used to coordinate other services: Firstly, it activates and deactivates user subscriptions according to user presence status and defined preferences. Secondly, it manages content adaptation to the characteristics of the applied user device. The component cooperates with personal mobility management, user profile, and content adaptation component, and relies on the publish/subscribe middleware for receiving and disseminating information relevant to service

coordination.

The *personal mobility management* component is responsible for maintaining up-to-date information about the current subscriber's presence mode. It defines a mapping of a unique user identifier to an end communication point where the subscriber is currently reachable. An end communication point identifies uniquely the terminal and the application that is used for receiving the content. It can be extended to track and store the user's geographical position for location-based services as described in [36]. The detailed description of the personal mobility management component is given in Section 6.4.

The *user profile* component stores and manages user profiles and enables a subscriber to define rules to customize the service. It stores user's subscriptions and the information about subscriber's communication points. It is closely related to the personal mobility management component because it stores the information about subscriber's default communication points such as e-mail addresses, or mobile phone numbers, that represent an active communication point stored by the personal mobility management component.

A subscriber can decide which subscriptions apply to a particular end communication point, current location, or time of day. Content can thus be queued for later delivery to a suitable device according to user preferences. A user profile can contain device capability data following the Composite Capability/Preference Profile (CC/PP) recommendation [28]. There are already a number of solutions that define the type of data that are stored in user profiles. Examples are *preference registry* designed within the ICEBERG project [122], and the "data recharging" profiles that evaluate the utility of certain content for a specific user [26]. The open problems are related to security and privacy: will the profile be stored on user devices, or will a broker store a copy, and who can access and change a user profile.

Content adaptation deals with the problem of client and network variability in mobile environments. Data compression and data conversion are standard techniques for client and network variability adaptation [81]. For example, an image must be transformed into a new format to be displayed on a mobile phone, or a smaller and lower quality image is sent over a low-bandwidth connection. Dynamic adaptation [11] can be used for content dissemination services: The system monitors the environment, and acts upon changes, such as low bandwidth, or battery consumption. The publish/subscribe middleware can be used for distributing the data about environment changes.

6.2.3 Application Layer

The application layer contains components that are specific to a particular content dissemination service: It is affected by the type of content that is distributed to subscribers, and the particular application purpose. The *content management* component enables a publisher to define the channels, and create and manage device-dependent content which will be published on different channels. The *presentation* component is responsible for device-dependent content representation: The content must be adjusted to applied terminals in order to suit different display sizes and deal with terminal input

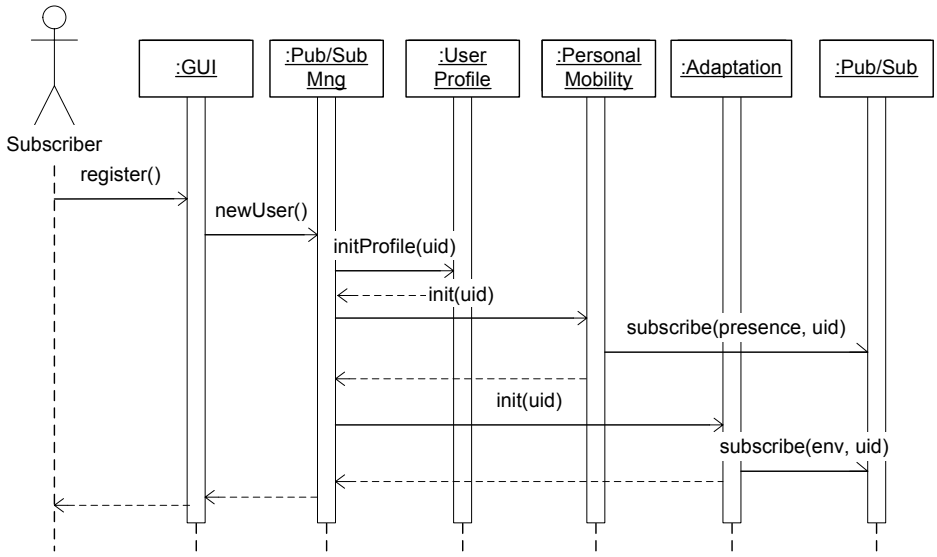

Figure 6.5: Registration of a new subscriber (UML sequence diagram)

limitations. Currently, XML and related technologies are used to create and manage flexible user interfaces [65]. The presentation-related problems, such as content structuring and partitioning, and simple input techniques are still open research topics.

6.2.4 Component Interaction

We show the interaction between the components of the proposed reference architecture using Unified Modeling Language (UML) sequence diagrams [17] that describe the following use cases: registration of a new subscriber in the system, subscription update due to subscriber disconnection, and the process of subscribing and content publishing.

The sequence diagram in Figure 6.5 depicts the component interaction when a new subscriber registers with the system for the first time. The subscriber uses a graphical interface (GUI) to define his/her user profile. GUI directs the request for registration of a new user to the publish/subscribe management component (Pub/Sub Mng): Firstly, Pub/Sub Mng initializes a user profile with a unique user identifier (uid) using the data provided by the subscriber. Secondly, it registers the user with the personal mobility management component (Personal Mobility). This component uses the publish/subscribe infrastructure to receive notifications regarding user connections to, and disconnections from the system: It subscribes to a special channel, presence, using uid as a filtering constraint. Each time the subscriber connects to or disconnects from the system, the relevant information will be published on the channels connect and disconnect. Therefore, the Personal Mobility component will be able to react to such occurrences and update the subscriber's presence status. Thirdly, Pub/Sub Mng contacts the adaptation component (Adaptation) that in turn subscribes to a special channel env that transports the information about the changes in service environment related to the user perception of

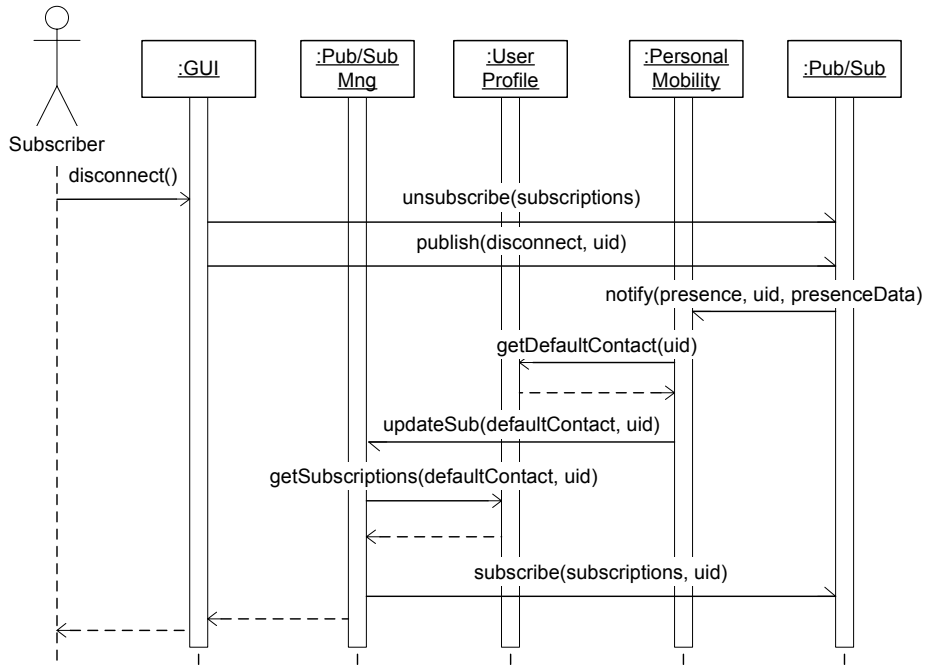

Figure 6.6: Subscription update due to disconnection (UML sequence diagram)

the service.

The sequence diagram in Figure 6.6 presents a scenario that causes subscription update due to subscriber disconnection. We assume that the subscriber has defined a default communication point that is stored in the user profile. When the subscriber decides to close an application used for receiving the published content (GUI), GUI sends an unsubscription request to the publish/subscribe middleware (Pub/Sub) and notifies a Personal Mobility component about user disconnection through the presence channel. The Personal Mobility component requests the data about the default user communication point from the User Profile component, and informs the Pub/Sub Mng about the new subscriber status. To activate subscriptions that the subscriber has chosen as valid in case of disconnection, the Pub/Sub Mng first retrieves the information about default subscriptions from User Profile, and subsequently activates the default subscriptions.

The sequence diagram in Figure 6.7 shows the component interaction for two representative use cases: publish (a publisher releases content to a channel) and subscribe (a subscriber subscribes to the channel). We assume that a subscriber uses a special application for receiving the published content (Receiver) that is independent of the GUI for defining and modifying subscriptions. The subscriber sends the subscription request from GUI to the Pub/Sub Mng component which in turn updates the user profile. When the user activates a Receiver that is used for receiving the content, the receiver must first get the information about valid subscriptions from User Profile, and activate the valid subscription by sending a subscribe request to the Pub/Sub component.

To publish content, the publisher defines a message using a GUI which updates the content storage

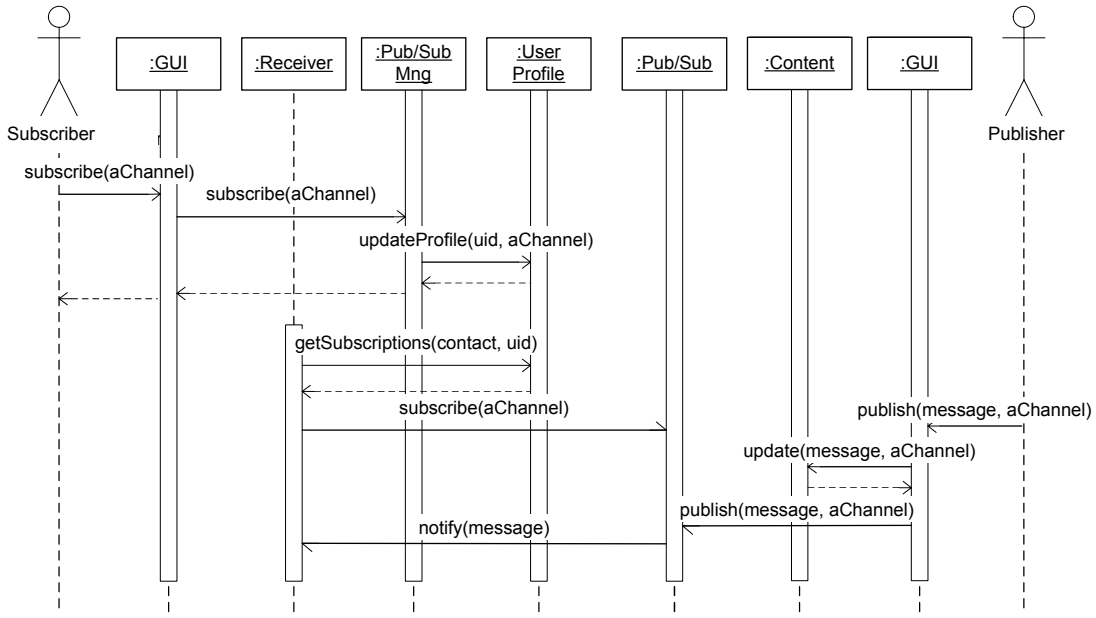

Figure 6.7: Sequence diagram for publish and subscribe use cases

of the Content component, and chooses a channel on which the message should be published. The GUI submits the published message to the Pub/Sub, and Pub/Sub notifies the receiver about the published message.

6.3 Publish/Subscribe as a Mobile Web Service

We propose the design of a Web-based publish/subscribe component which implements the communication layer of the proposed reference architecture [93]. The proposed component facilitates the implementation of content dissemination services for mobile users who apply various types of terminals for content receipt: It offers a general set of methods for the implementation of publish/subscribe-based interaction and uses other components, such as mail, SMS, MMS, or JMS for the actual content transmission. The service is mobility-enabled: The components used for content transmission are applicable in mobile scenarios and facilitate the receipt of published content on various subscriber's devices.

Web service. Although publish/subscribe interaction style has been recognized as a valuable service in a number of application domains, no Web services that offer the generic publish/subscribe functionality are available so far. We argue that publish/subscribe is a generic service that is required in various application domains such as content dissemination, notification services, instant messaging, or groupware and collaboration systems. Publish/subscribe has indeed been recognized as one of the basic services in the Web services architecture for groupware systems proposed in [38].

Web service, an emerging model for distributed computing on the Web, can be regarded as soft-

ware system providing a well-described functionality accessible over the network. Service description specifies its interface in a machine-processable format and facilitates the interaction between various Web services to enable their integration in order to provide more complex value-added services [37, 121]. Web services are designed as self-contained software components that can be published, discovered, and invoked over the Internet. They apply XML-based standards for the transport of messages and service description. Simple Object Access Protocol (SOAP) [120] is used as the communication protocol for invoking service methods and conveying processing results. Web Services Description Language (WSDL) [119] is an XML-based language used for describing Web services.

The publish/subscribe systems that are in use today offer a proprietary set of interfaces and APIs for integration into other systems which burdens an application programmer with the details of each specific implementation. Therefore, we design a generic publish/subscribe component as a Web service to facilitate simple and efficient integration of publish/subscribe functionality into other systems. It generalizes the common publish/subscribe constructs and can be regarded as a layer above the existing publish/subscribe infrastructures.

A Web-based publish/subscribe service should offer a set of basic services for publishing, subscribing, and creating channels for content classification. It should be remotely accessible, and accept XML messages that define the information needed to perform the requested functionality. We assume that the generic publish/subscribe service will rely on a number of other components, such as mail, SMS, MMS, JMS, or other publish/subscribe middleware components that will perform the actual transport of the published content.

6.3.1 Architecture

Figure 6.8 depicts the architecture of the proposed Web-based publish/subscribe service. We employ the layered approach in which the publish/subscribe service is an intermediary between other services that require publish/subscribe, such as content dissemination services, instant messaging, or groupware services, and specific components that provide the transport of data for the publish/subscribe service. The publish/subscribe service offers a well-defined interface and generalized means to invoke the dissemination services regardless of the actual transport mechanism provided by, for example, mail, SMS, MMS, or JMS component. It enables a user of the publish/subscribe service to specify the preferred transport mechanism for the particular request. Further on, it is open and extendible by available transport components suitable for publish/subscribe content dissemination.

The Web-based publish/subscribe service implements the communication layer of the reference architecture and is used as a binding between higher-level services of the service layer and components providing publish/subscribe communication. Figure 6.9 shows the correspondence between the reference architecture and the proposed Web-based publish/subscribe service.

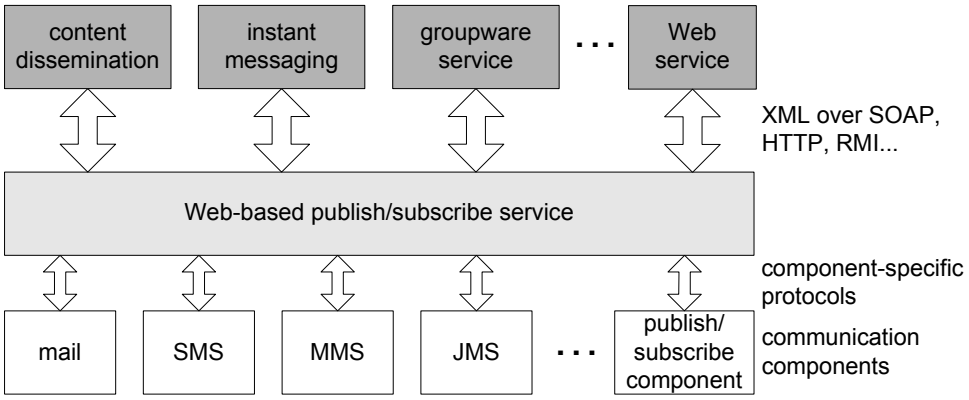

Figure 6.8: Web-based publish/subscribe service

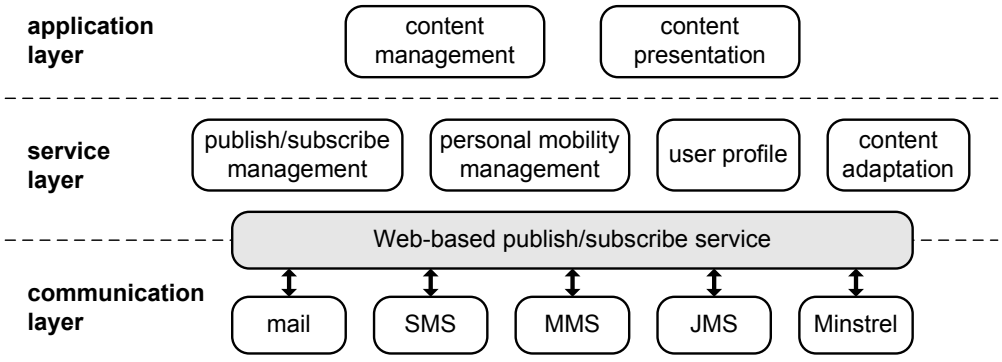

Figure 6.9: Web-based publish/subscribe service with respect to reference architecture

Table 6.2: Functionality offered by the publish/subscribe Web service

`publish`	Publishing the content on a channel using a defined transport component.
`subscribe`	Subscribing to a channel with a specified preferred mechanism of content receipt for this subscription.
`unsubscribe`	Unsubscribing from a channel.
`createChannel`	Defining a new channel for content classification with a predefined mechanism for content delivery.
`deleteChannel`	Deleting a specified channel.

6.3.2 Service Interface

There are two basic functions a publish/subscribe content dissemination service needs to provide: *publishing* and *subscribing*. *Channel definition and creation* is needed since the content is published on a channel, and a subscription is to a channel. Therefore, we propose a simple interface which offers the methods `publish`, `subscribe`, `unsubscribe`, `createChannel`, and `deleteChannel` listed and defined in Table 6.2. We find the listed methods sufficient for the publish/subscribe-based interaction implementation. These methods are requested in pull-style by submitting an XML message to the publish/subscribe service. On the other hand, the process of receiving the published content is independent from the generic publish/subscribe service: It is performed by a receiving process, for example, a mail reader or a JMS receiver, running on a subscriber's terminal. The receiving process must be transport specific since the notification is always sent in push-style through a specific transport component without an intermediary.

XML messages carry the parameters needed to perform the requested action: The mandatory parameter for each request is the information about the transport component that will subsequently perform content delivery. Figure 6.10 shows an example XML message that requests the creation of a new weather channel. The parameters that are needed to create a new channel are the information about the channel creator and the definition of the new channel. The channel creator in the example is a user, but it is possible that another Web service sends such a request. The channel is defined by it's name, unique id, and an URL that uniquely identifies the transport component and it's channel. For example, `jms://aloha.tel.fer.hr/topic=weather` specifies that JMS is used for content dissemination, `aloha.tel.fer.hr` is the name of the JMS server host, and `weather` is the name of the JMS topic.

Figure 6.11 shows an example XML message that requests subscription to an existing channel. The required parameters are subscriber's contact information and a subscription. The subscriber's contact information uniquely identifies the subscriber and can be obtained from a user profile component. The subscription is composed of the channel data and an XPath expression refining the subscription. XPath is a language for addressing parts of an XML document [118]: It offers simple and expressive

```
<?xml version="1.0"?>
   <ps:createChannel xmlns:ps="http://www.tel.fer.hr/webservices/pubsub/">
      <ps:creator>
         <ps:user name="Ivana Podnar"
                  email="ivana.podnar@fer.hr"
                  mobile="+3859991234567"
                  id="unique_id_for_ivana" />
      </ps:creator>
      <ps:channel name="Weather channel"
                  url="jms://aloha.tel.fer.hr/topic=weather" />
   </ps:createChannel>
```

Figure 6.10: An example XML message requesting channel creation

```
<?xml version="1.0"?>
   <ps:subscribe xmlns:ps="http://www.tel.fer.hr/webservices/pubsub/">
      <ps:subscriber>
         <ps:user name="Ivana Podnar"
                  email="ivana.podnar@fer.hr"
                  mobile="+3859991234567"
                  id="unique_id_for_ivana" />
      </ps:subscriber>
      <ps:subscription>
         <ps:channel name="Weather channel"
                     url="jms://aloha.tel.fer.hr/topic=weather"
                     id="unique_id_for_jms_weather_topic" />
         <ps:xpath>
            //content/@country="Croatia"
         </ps:xpath>
      </ps:subscription>
   </ps:subscribe>
```

Figure 6.11: An example XML message requesting subscription to a channel

means to specify and select the elements and attributes of an XML document, and can therefore be used to express XML document filters. XPath can be used for expressing content-based subscriptions, and XPath engines can be used to decide whether a published XML document satisfies the XPath subscription. Algorithms and techniques for efficient filtering of a large number of XML documents using XPath have been proposed in [8, 46].

The example XML message in Figure 6.12 initiates the process of publishing the content on an existing channel. The parameters are publisher's data, information about a channel, and content definition. Content definition in the given example consists of a name, a timestamp, validity dates, an optional short text message, and a URL. Content definition carries the notification, not the actual content, since it is preferable that the data that is pushed to a terminal, especially in mobile scenarios, is concise. A subscriber can request the actual content using the provided URL.

The publish/subscribe component uses the information included in the presented XML messages for submitting the request further to a specific transport component. The main task of the publish/subscribe service is the mapping of standard XML messages to transport-specific method invocations. Some transport components are inherently publish/subscribe-enabled: For example, JMS incorporates the principles of publish/subscribe which simplifies the process of mapping an incoming XML

```
<?xml version="1.0"?>
   <ps:publish xmlns:ps="http://www.tel.fer.hr/webservices/pubsub/">
      <ps:publisher>
         <ps:user name="Ivana Podnar"
                  email="ivana.podnar@fer.hr"
                  mobile="+3859991234567"
                  id="unique_id_for_ivana" />
      </ps:publisher>
      <ps:channel name="Weather channel"
                  url="jms://aloha.tel.fer.hr/topic=weather"
                  id="unique_id_for_jms_weather_topic" />
      <ps:content name="Forecast for September 06 2003"
                  url="http://aloha.tel.fer.hr/weather/hr/2003/sept/forecast060903.html"
                  timestamp="Fri, Sept 05 10:44:04 CEST 2003"
                  validFrom="05.09.2003"
                  validTo="07.09.2003"
                  id="unique_id_for_content"
                  country="Croatia" >
   </ps:publish>
```

Figure 6.12: An example XML message initiating content publishing

request to a JMS-specific request. Mail, SMS, and MMS do not have the built-in publish/subscribe constructs: The publish/subscribe component needs to extend the basic operation of such components to enable their application in publish/subscribe scenarios. For example, the publish/subscribe component is responsible for defining mail messages containing the published content and for submitting them to the mail server for subsequent delivery to interested subscribers. It performs the tasks similar to mailing lists: It maintains a list of subscribers to channels, and on top of simple channel categorization it can perform mail filtering based on XPath expressions. The difference between mailing lists and this approach is in the filtering part: The list of receiving mail addresses is dynamically determined for each published notification.

The described Web-based publish/subscribe service does not directly address problems related to mobility. Mobility is facilitated by transport components that are mobility-aware and facilitate the receipt of published content on various devices. Moreover, the combination of various transport components offers usage flexibility: For example, a user may apply mail for receiving notifications while having a permanent Internet connection, and receive SMS notifications while on the move.

6.4 Personal Mobility Management

The personal mobility management (PMM) service stores, updates, and distributes the user's presence and contact information. The presence information describes the current user communication capabilities and preferences with respect to the applied terminal, application, and user state. We assume that a subscriber's presence information is updated as the subscriber starts and uses various applications that may update the presence information, such as an instant messaging client, or a publish/subscribe content receiver. Furthermore, a subscriber can define a default communication point which is activated in case the current presence information is unavailable.

Table 6.3: Communication point definitions and examples

`mail://username@host`	`mail://ivana.podnar@fer.hr`
`sms://+(phone_number)`	`sms://+3859991234567`
`mms://+(phone_number)`	`mms://+3859991234567`
`scheme://host:port/ application_parameters`	`jms://receiver.tel.fer.hr:8738/ m-NewsBoardReceiver` `minstrel://receiver.tel.fer.hr:9090/ minstrel/Receiver`

We define the presence information in accordance with the 3GPP presence service specification [3]. The presence information includes the identifier of an end communication point, user's status, and user's terminal.

Communication point. The end communication point provides the information about the type of the service or application that can be used for communicating with a user, e.g., SMS, MMS, e-mail, instant messaging service, publish/subscribe or push-based content receiver; and the contact address that is needed to carry out the communication. A communication point can be defined in the form of an URL: Table 6.3 lists examples of URLs that can be used to uniquely identify and describe end communication points.

User's status. The user's status field defines whether a user is willing and capable to accept the requested communication. Examples of user's status are *available*, *discreet*, or *unavailable*. Since content delivery does not require immediate user's attention and involvement in communication, user's status is not vital for content dissemination services.

Terminal's status. The information about the terminal's current status can also be included into the presence information. Possible status values are *on-line*/*off-line* for computers, or *busy*/*idle*/*detached* for mobile phones.

The presence information can also include the information about the terminal's geographical location required by location-based services.

The changes of user's presence information need to be reported to the PMM server that updates the user's presence status. The publish/subscribe event-based approach is a natural communication mechanism for updating the presence information: As soon as the status changes, the change is reported to the PMM server. In publish/subscribe terms, the PMM server is a subscriber to a special channel/topic, e.g., `presence`, that is used for distributing the presence data. All other services that require the presence information can query the PMM server in the standard request/reply style to obtain the presence data, or may become subscribers to the presence channel/topic. Publishers of presence data are various applications that may update the presence information. For example, when

a user starts a JMS content receiver, the application can declare the JMS receiver as the end communication point. By closing the receiver, a new message canceling the active JMS presence data would be published. In case no other presence data is available, the PMM server can activate a user's default communication point, e.g., e-mail or SMS.

The user's presence data interferes with user's privacy: The update and retrieval of this information must therefore be secured and authenticated.

Presence update. The presence data can be updated only in case proper user credentials are provided to avoid situations in which a malicious user impersonates the user. We propose that the published presence data carry user credentials that can be verified by the PMM. For example, a unique user identifier and a password can be transported with the presence data and a time-to-live period over a secure communication protocol such as Secure Sockets Layer (SSL). It is also possible to use a public key distribution similar to the approach presented in [4]: Apart from being identified by a unique identifier (Id_u), each user has a private/public key pair (D_u/E_u). When the user registers with the PMM server for the first time, it provides its Id_u and E_u. When reporting the change of presence data to the PMM, the user sends a tuple consisting of Id_u, the encrypted presence data $D_u(Pres_u)$, an expiry field for the new presence mapping T_u, and a signature of the new mapping, i.e., $D_u(Id_u, Pres_u, T_u)$. The PMM server uses the user's public key E_u to check the signature by verifying that $E_u(D_u(Id_u, Pres_u, T_u)).Id_u = Id_u$ which confirms that the update comes from the user. The presence data is obtained by decrypting $D_u(Pres_u)$, i.e., $E_u(D_u(Pres_u))$. The presence data is valid until T_u expires, and must be renewed. Otherwise, the default presence status is activated. A possible problem with this approach is that a user applies various devices for updating the presence data. The private/public key pair and the Id_u must therefore be stored on a special smart card and transfered to the applied terminal.

Presence retrieval. Only the parties that satisfy the authorization policies can access the user's presence information. The PMM server administrator can allow access to the presence data only to authorized parties that provide adequate credentials. For example, if a PMM server uses a JMS queue or a JMS topic for distributing the presence information, the JMS administrator must allow access to the JMS presence queue/topic only to parties that are authorized to receive the user's presence data.

Chapter 7

m-NewsBoard: A Case Study

This chapter presents m-NewsBoard, a news dissemination service for mobile users based on the publish/subscribe interaction model. This service enables users to publish and receive news of their interest, and yet stay mobile. Users can browse the repository of current news on a WAP-enabled mobile phone, or in a desktop browser, and publish their news using the m-NewsBoard's Web interface. In addition, they may subscribe to particular news categories, and supply keywords to refine their subscriptions. Subscribers will receive either e-mail or JMS notifications when news matching their subscriptions are published. m-NewsBoard is a prototype system implemented using the loosely-coupled remotely accessible services that have been identified as parts of the service-oriented reference architecture presented in Chapter 6. The implemented components, the Web-based publish/subscribe service, and the personal mobility component in particular, offer generic functionality and are applicable for integration into various content dissemination services.

The chapter is structured as follows: Section 7.1 presents m-NewsBoard's implementation. Firstly, in Section 7.1.1 we describe the usage scenarios to show the user's perspective of the application. Secondly, we present the system architecture, compare it to the proposed reference architecture, and discuss m-NewsBoard's implementation details in Section 7.1.2. The detailed description of the Web-based publish/subscribe service prototype implementation is presented in Section 7.2. Section 7.3 offers a solution for the device independent mobility-aware content receipt, and personal mobility management. We discuss the characteristics of the implemented system in Section 7.4, and contrast them to the general content dissemination service requirements that have been identified in Section 6.1.

7.1 m-NewsBoard - a News Dissemination Service

m-NewsBoard is a content dissemination service for publishing and delivering news in the form of multimedia messages to mobile users [95]. It offers flexible usage scenarios enabling personal mobility: Users can apply various devices for browsing, publishing, and receiving the news. For example, users may browse the repository of published and unexpired news, define and publish their news, and define subscriptions using a WAP-enabled mobile phone, or a desktop browser. They may subscribe

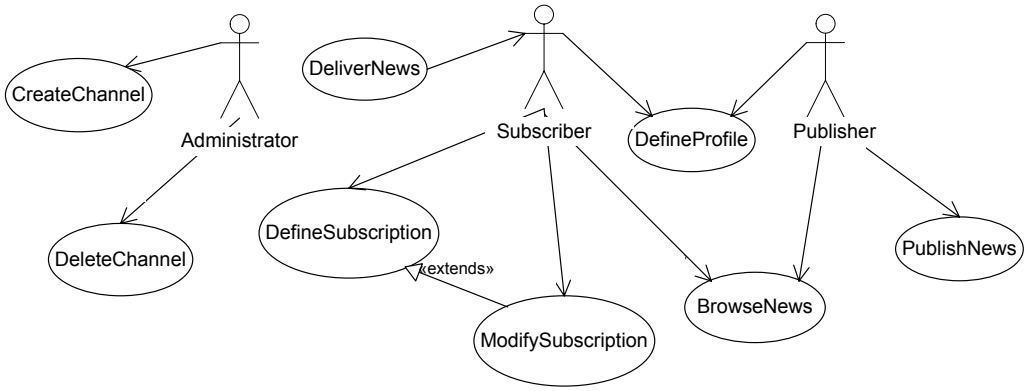

Figure 7.1: m-NewsBoard use cases

to particular news categories, supply keywords for further specialization of their subscription, and choose the preferred means for receiving the news at the time of news publishing.

7.1.1 Usage Scenarios

The UML use case diagram in Figure 7.1 defines system usage scenarios. There are three types of system users: administrator, publisher, and subscriber. An administrator has privileges to administer user profiles of other users, and to create and modify channels that are used for content classification. A publisher can publish news on existing channels and browse the repository of published news in pull-style. A subscriber can also browse the news repository, and additionally, actively receive notifications about news publications according to defined subscriptions. New system users have to register when they use m-NewsBoard for the first time, and provide the data needed to create a new user profile.

m-NewsBoard supports the following use cases:

CreateChannel, DeleteChannel. An administrator can create a new channel by providing its name, a short description, and the delivery method used for transporting the content to subscribers. The administrator can also invalidate an existing channel.

DefineProfile. Each system user is associated with a user profile. A user defines the profile when it first registers with the service and provides the following data: user name, password, full name, e-mail address, and mobile phone number.

DefineSubscription, ModifySubscription. A subscriber defines the subscription to an existing channel and optionally provides a list of keywords for further filtering of news published on the subscribed channel. For specifying the subscription, a user is presented with a screen shown in Figure 7.2. By choosing a channel, the subscriber also chooses the delivery method for notifications. Note that a subscriber can subscribe to the “same” channel, i.e., the channel that delivers

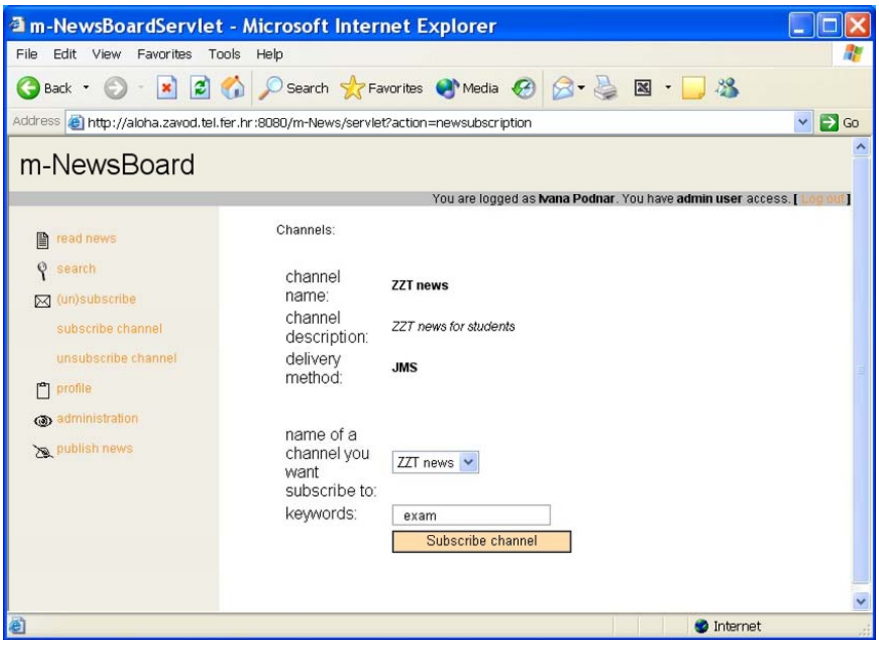

Figure 7.2: Subscribing to a channel

the same content, multiple times. For example, the subscriber may subscribe to the channel ZZT twice, using JMS and e-mail as means for content receipt. In such a case, the subscriber will receive notifications in a JMS receiver application while it is active, and the service will activate e-mail subscription while the subscriber is not using the JMS receiver. Personal mobility management enables tracking of user's presence information.

PublishNews. A publisher may publish news on an existing channel. He/She defines a news title, news body, the date of news expiry, chooses a channel that classifies the news, and optionally provides an additional file to be published with the news and a list of keywords that further describe the news. Keywords are used for content-based news filtering. The added file will be stored on the m-NewsBoard's Web server, but it will not be included in notifications that are sent to subscribers. Subscribers will receive the URL of the published file to retrieve the document. Figure 7.3 shows the m-NewsBoard's publishing screen.

BrowseNews. A registered user can browse the repository of valid news that are stored on the m-NewsBoard's Web server: This is the traditional pull-based operation offered by the m-NewsBoard application. The list of news is created on the fly for each request with unexpired news from the news repository depending on the type of used browser. Figure 7.4 shows the list of published news presented in a desktop browser.

DeliverNews. This use case enables a subscriber to receive notifications about news publication in case the published news corresponds to his/her subscription. A subscriber that uses JMS for

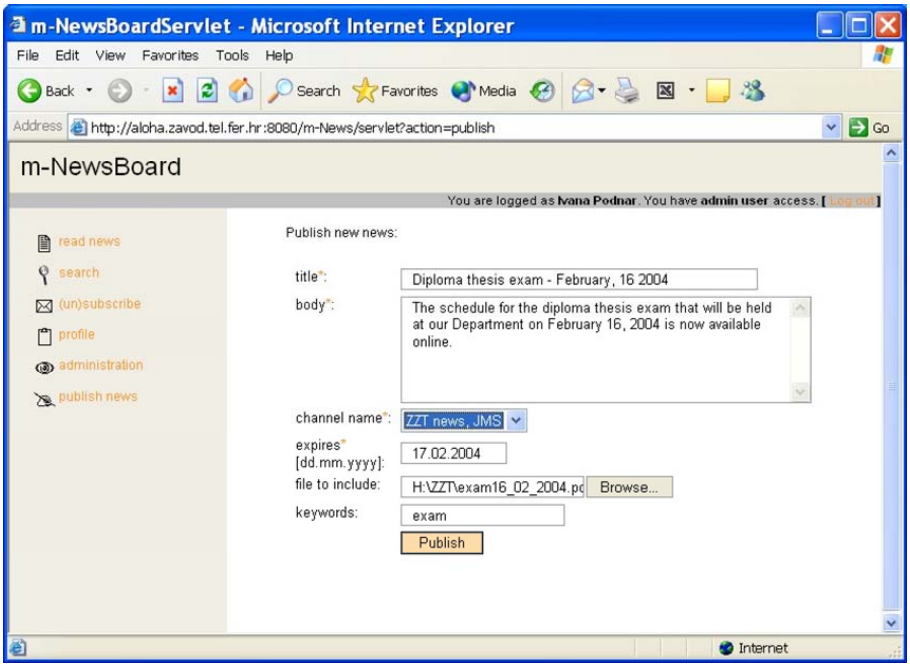

Figure 7.3: Publishing news on a channel

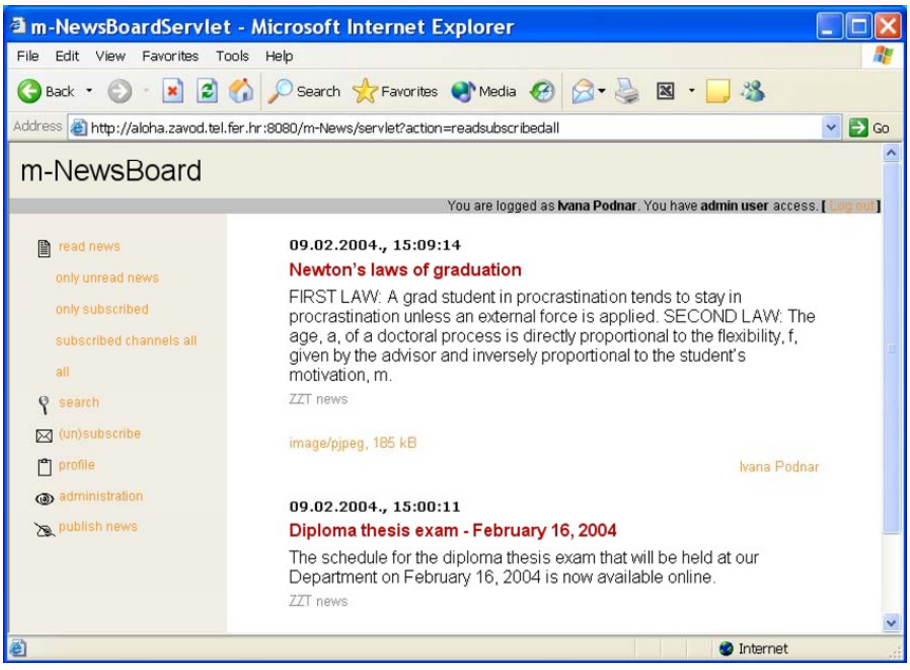

Figure 7.4: Reading the published news in a desktop browser

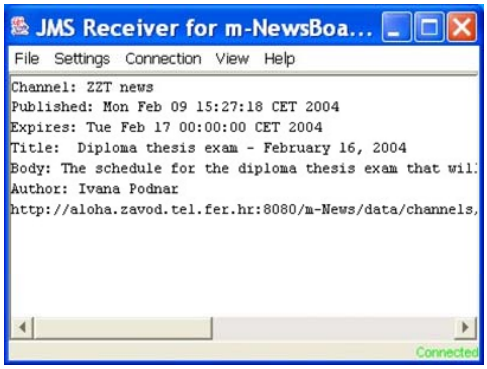

Figure 7.5: The published message in a JMS desktop receiver

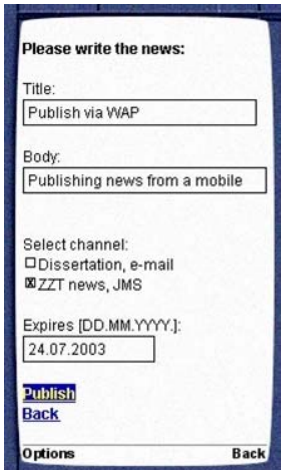

Figure 7.6: Publishing screen on a mobile phone

news receipt would receive a notification in the active JMS receiver at the time of news publishing as shown in Figure 7.5.

m-NewsBoard's implementation supports WAP-enabled terminals: The listed and presented use cases are also supported if users apply WAP-enabled mobile phones. For example, Figure 7.6 shows the publishing screen in a mobile phone simulator. The presented screen is fitted to show all the contents of the publishing screen for representation reasons only: This screen would be split into several screens on a real mobile phone.

7.1.2 Description of System Implementation

The architecture of the m-NewsBoard system consists of a WAP-enabled Web interface that interacts with the Web-based publish/subscribe service and a personal mobility component as depicted in Figure 7.7. The Web interface is responsible for news presentation: It interprets the incoming requests,

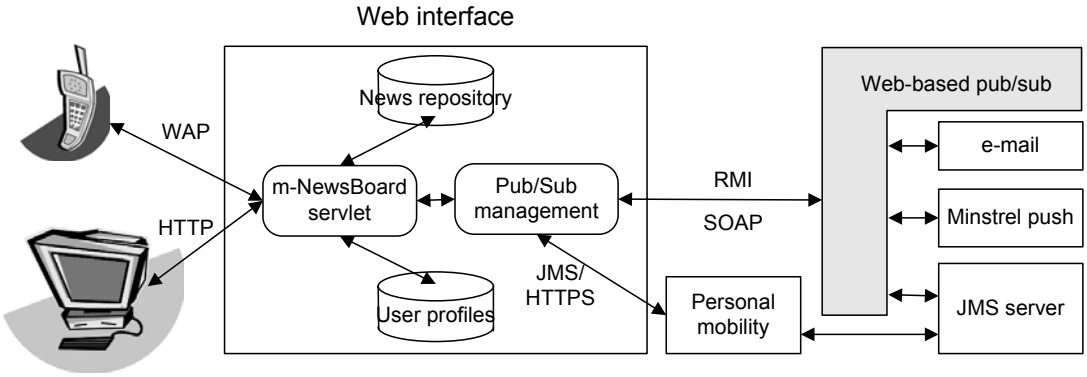

Figure 7.7: m-NewsBoard architecture

updates the news repository, and submits the publish/subscribe requests to the publish/subscribe service which is in charge of news dissemination according to user subscriptions. The Web interface, i.e, its publish/subscribe management component, interacts with the personal mobility component that maintains the presence information about registered users, and enables modifications of user subscriptions depending on the presence data.

The WAP-enabled Web interface is implemented as a Java servlet which runs within a Web server and supports both HTTP and WAP. It maintains a repository of published news, and stores user profiles that contain user subscriptions and describe user access rights regarding the service. The servlet provides the service graphical user interface: It is designed following the principle of clear separation of content and layout. The content is stored in the form of XML documents, while special HTTP and WAP templates define the layout. The servlet incorporates a publish/subscribe management component that invokes the services offered by the Web-based publish/subscribe service, and interacts with the personal mobility component to receive the relevant presence information that will modify active subscriber's subscriptions. A detailed description of the m-NewsBoard servlet implementation can be found in [97].

If we compare the m-NewsBoard's architecture to the reference architecture depicted in Figure 6.4, it can be concluded that the m-NewsBoard servlet implements content management and content presentation. Furthermore, it stores user profiles and incorporates a solution for publish/subscribe management. The publish/subscribe communication layer is implemented as a stand-alone Web-based service providing publish/subscribe communication primitives. Section 7.2 gives the detailed description of the Web-based publish/subscribe service implementation. The prototype implementation of the personal mobility component offers a centralized solution for maintaining subscriber's presence data. It is further described in Section 7.3. The m-NewsBoard's current implementation provides no support for content adaptation: The content structure is determined by publishers who decide what type of content will be sent on a particular channel depending on the applied transport mechanism, such as e-mail or JMS.

The m-NewsBoard's system architecture is truly distributed as shown in the deployment dia-

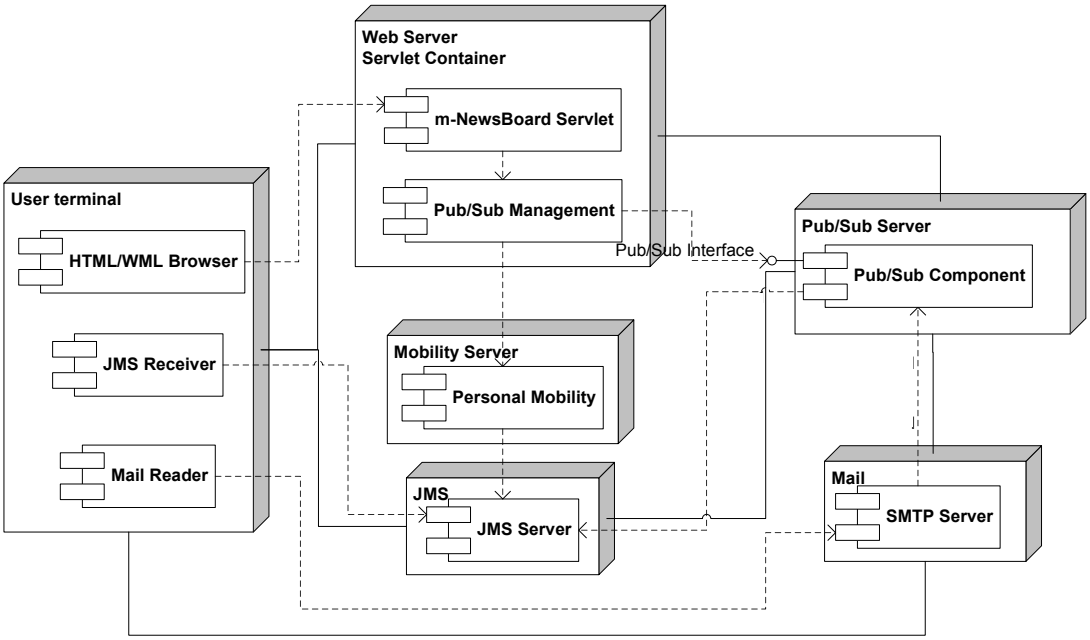

Figure 7.8: Deployment diagram

gram in Figure 7.8. User interacts with the Web interface using an HTML or WML browser and the communication between the m-NewsBoard servlet and the browser is performed over HTTP or WAP. The web servlet processes user requests and 'translates' them into Java Remote Method Invocation (RMI) calls [110] or Simple Object Access Protocol (SOAP) messages [120] because the publish/subscribe service offers an interface that is remotely accessible using either Java RMI, or SOAP. The publish/subscribe service processes each request, and submits it further to specialized delivery components such as an SMTP server, or a JMS server. The personal mobility component and the publish/subscribe management component communicate using JMS over HTTPS. A user employs transport-specific applications such as a JMS receiver, or an e-mail reader, for receiving notifications about news publication.

We show the interaction between the m-NewsBoard components in the sequence diagram that is depicted in Figure 7.9. The sequence diagram illustrates the processes of subscribing and publishing news using JMS as transport protocol. A subscriber uses a Web browser to subscribe to a channel and a JMS receiver for reading the published news. Note that the receiver is a lightweight component that can run on a desktop computer, a handheld computer, a mobile phone, or in a Web applet: It performs a single function of displaying the received news. A publisher uses a browser to publish news on a defined channel.

Subscribing. In the example scenario we assume that a publisher or system administrator has defined a news channel, and that the corresponding JMS topic exists on the JMS server. Firstly, a subscriber defines his/her subscription which consists of a preferred means for receiving the news,

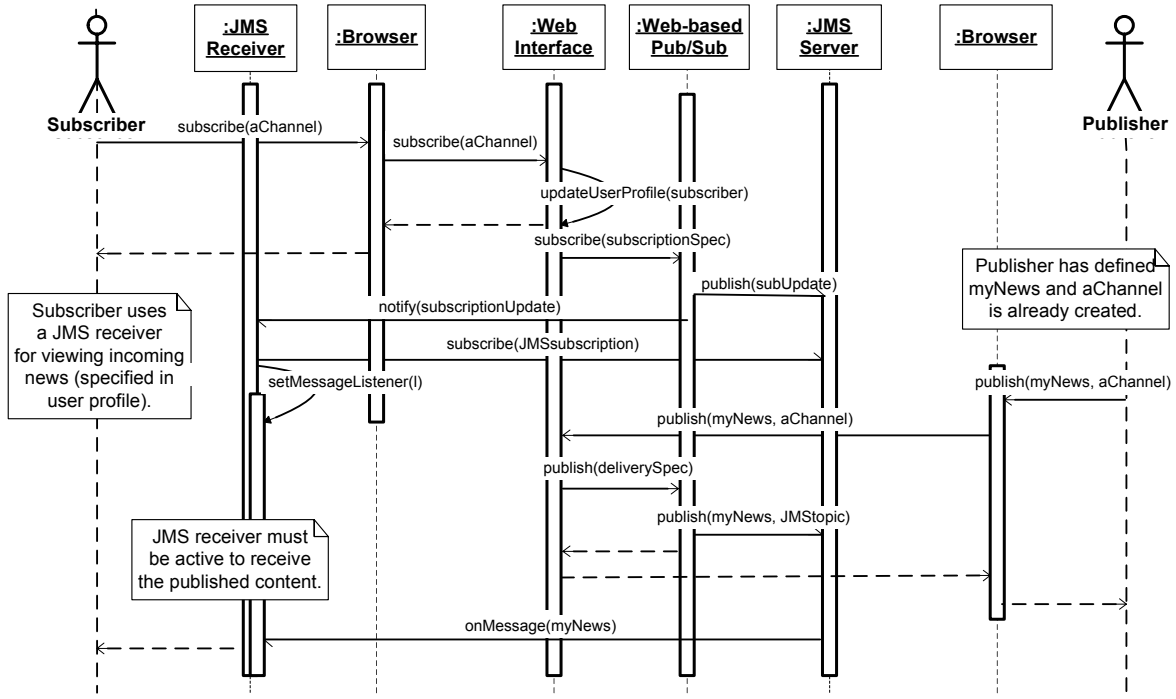

Figure 7.9: Sequence diagram that shows the interaction between NewsBoard's components.

the channel identifier, and an optional list of keywords for news filtering. In the example scenario the subscriber selects JMS as the preferred means for news receipt. Secondly, the servlet processes the subscription request and updates the subscriber's user profile by adding the new channel to his/her subscription. Thirdly, the servlet will relay the subscription request to the dissemination component with the specific parameters that define the user's subscription, i.e., JMS as the delivery component, the subscriber's id, the channel id, and the list of keywords.

Subsequently, the dissemination component needs to initiate and register the new subscription with the JMS server. However, the problem is that the dissemination component is not the receiver of published messages, instead it is a JMS receiver used by the subscriber. The JMS receiver must create a new message listener for the subscribed JMS topic that can accept incoming messages from the JMS server. Since the JMS receiver has no knowledge about user subscriptions that are processed through the servlet, the dissemination component needs to notify the receiver about changes in user subscriptions. We have decided to implement the interaction between the JMS receiver and the dissemination component through the JMS server. During the JMS receiver bootstrap, the receiver subscribes to a special JMS topic - `subscriptionUpdate` - with a filter requesting that the user id of the subscription update message matches the subscriber id. Accordingly, the dissemination component publishes the data about the changes in user subscriptions through the `subscriptionUpdate` topic specifying whether a subscription or unsubscription request has been processed, the subscriber's id, the JMS topic name, and the list of keywords if such have been defined. In the example sequence

diagram in Figure 7.9, the dissemination component publishes a message about the new subscription defined by the subscriber, and the running receiver receives the subscription update message through the JMS server. Consequently, the receiver sends a subscription request to the JMS server and starts a new message listener for the specified topic.

Publishing. The process of publishing is performed using a browser: A publisher specifies the news data and decides on which channel it should be published. The servlet will process the request, store the news in its repository, and transfer the publishing request to the dissemination component. The dissemination component submits it further to the JMS server which finally disseminates the news to all subscribers of the particular JMS topic. Eventually, the subscriber's JMS receiver will receive the published news and notify the subscriber that a news of interest has been published.

7.2 Publish/Subscribe Service Implementation

The publish/subscribe service implements a Web-based interface defined in Section 6.3 using Java RMI and SOAP. Each method of the defined interface accepts an XML message describing the request as an input method parameter. We have incorporated two transport components as a proof of concept implementation: e-mail and JMS. Each request identifies the transport component that is used to complete the actual request. The main task of the Web-based publish/subscribe component is to interpret an incoming request, and map it to the specific requirements and format of the transport component.

JMS offers publish/subscribe constructs that simplify the mapping process. E-mail, on the other hand, does not have the built-in publish/subscribe functionality: It can be used to transport the content, and the specific publish/subscribe functionality is added as part of the publish/subscribe service implementation. We outline the mapping between incoming requests that are processed by the publish/subscribe Web service and component specific method invocations in Table 7.1. The incoming request `createChannel` that specifies JMS as the transport component creates a new JMS topic publisher. For example, a request with the URL `jms://aloha.tel.fer.hr/topic=ZZTnews` creates a new topic `ZZTnews` on the JMS server running on the host `aloha.tel.fer.hr`. The same request specifying e-mail as the transport component, e.g., `mail://ZZTnews@tel.fer.hr` creates a new mailing list that is maintained by the publish/subscribe Web service. The request `deleteChannel` closes an active JMS topic publisher, or removes an existing mailing list from the publish/subscribe service repository. The request `subscribe` that specifies JMS as the transport component creates a new durable topic subscriber. While a durable JMS subscriber is disconnected from the JMS server, the server stores messages for the subscriber. A durable subscriber is identified by a unique identifier: It is necessary to provide only the durable subscriber identifier to re-initiate subscriptions and receive messages published during disconnection. The request `subscribe` that chooses e-mail as the transport component adds an e-mail address to an existing mailing list. The request `unsubscribe` unsubscribes and removes an existing durable JMS subscriber from a JMS server, or removes a mail address from an existing mailing list maintained by the publish/subscribe

Table 7.1: Mapping publish/subscribe methods to JMS and e-mail specific implementations.

Publish/subscribe Web service	**JMS**	**e-mail**
`createChannel`	create JMS topic publisher	create new mailing list
`deleteChannel`	close JMS topic publisher	remove an existing mailing list
`subscribe`	create durable JMS topic subscriber	add subscriber to an existing mailing list
`unsubscribe`	remove durable JMS topic subscriber	remove subscriber from a mailing list
`publish`	publish message using JMS topic publisher	send e-mail message to mailing list members

service. The request `publish` will invoke either a publish method on an active JMS topic publisher, or initiate the process of sending messages to addresses of a specified mailing list.

Support for content-based subscriptions and message filtering is adjusted to specific component characteristics. Table 7.2 depicts a JMS representation of the published news as defined in Figure 7.3. JMS enables content-based message filtering through message selectors that are defined on JMS message properties. Every JMS message has standard properties such as JMSDeliveryMode, or JMSTimestamp, and application specific properties. We use application specific properties to define message selectors. For example, `keyword='exam'` is the message selector that is used to refine a subscription to the JMS topic `ZZTnews` defined in Figure 7.2. A published message that matches the defined subscription must include the additional JMS property `keyword` with the value `exam`. For e-mail, content-based subscriptions and message filtering need to be implemented additionally as part of the publish/subscribe service. We suggest the usage of XPath for defining message filters, e.g. `//content/@keyword="exam"`, and XPath processing tools, such as Xalan-Java [9], for matching published XML messages to XPath filters.

Here we describe the details of the Java RMI service implementation. Figure 7.10 shows a class diagram of the server-related classes and proxy classes used for communicating with specific transport components. Server-related classes are `DisseminationServer` that registers an RMI remote object, and `DisseminationImpl` implementing the remote object interface specified in `Dissemination`. Each request that complies with the interface definition is directed to a specific transport component, and a special proxy object, either an instance of `JMSProxy`, or `MailProxy`, handles the request and directs it to the appropriate component. Since the type of the instantiated proxy object is unknown prior to request receipt, we use the Factory Method pattern [53] that enables an object to instantiate an object whose type is specified at run time. In our solution, the `DisseminationImpl` instantiates `JMSProxy` and `MailProxy` objects using the class `DissProxyFactory` and the abstract class `DissProxy`. For the SOAP implementation the server-related classes are adjusted to implement a SOAP servlet. The service description using Web

Table 7.2: A JMS message representation

channel	ZZT news
name	Diploma thesis exam - February, 16 2004
id	`20040209115511_ivana`
valid from	1076281200000
valid to	1096063200000
type	content
size	1775435
MIME	application/pdf
URL	`http://aloha.zavod.tel.fer.hr:8080/m-News/data/channels/20040209115511_ivana/exam16_02_2004.pdf`
keyword	exam
JMSDeliveryMode	2
JMSMessageID	ID:113001
JMSPriority	4
JMSExpiration	1096063202406
JMSTimestamp	1076324113765
MessageBody	The schedule of the diploma thesis exam...

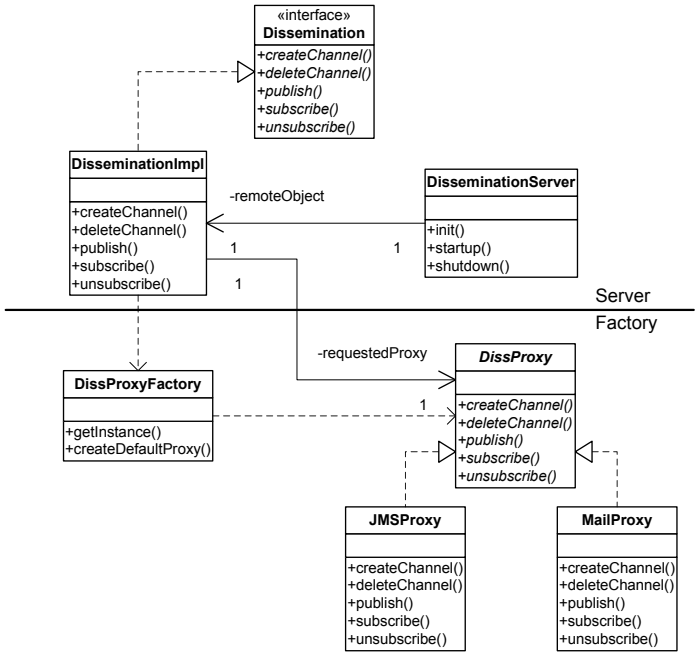

Figure 7.10: Class diagram of the Java RMI implementation

Services Description Language (WSDL) and a detailed description of the SOAP-based implementation can be found in [80].

7.3 A Solution for Personal Mobility

The solution for personal mobility relies on the JMS server functionalities: It uses special JMS queues for storing the presence information that is accessed on demand in the request/reply style, and JMS topics for distributing the presence information to active processes that are affected by changes in the subscriber's presence status. In the current implementation, a subscriber can either receive e-mail notifications, or JMS messages in an active JMS receiver. Therefore, the user's presence data can either contain a user's e-mail address, or a URL of a JMS receiver.

The process of news receipt is transport specific: A user needs a mail reader for receiving mail notifications, and a JMS receiver for JMS-based messages. The design of a JMS receiver is the most challenging in this context: We describe the design of the JMS receiver that is used for displaying the received messages in the m-NewsBoard application.

It is important to note that a user can specify a subscription to a single channel in three modes: e-mail delivery, JMS delivery, and combined e-mail/JMS delivery. The user presence information does not affect the e-mail based delivery. For the JMS-based delivery we use durable subscriptions to enable message storage during disconnections, and a special solution for storing and updating active subscriptions in device-independent style. The special solution is needed because a user can apply different devices for receiving the content, and no subscription information can be stored on a device. For the combined e-mail/JMS delivery we assume that JMS delivery is used while a user applies a JMS receiver, and e-mail is the default delivery mechanism in case when the JMS receiver is inactive.

Receiving the news using JMS. The JMS receiver offers a single functionality: It displays the messages for an authenticated user that comply with the defined user subscriptions. The receiver needs to be independent of the applied terminal: A subscriber may start a receiver on one terminal, even in an applet, receive the published messages, then disconnect and stop the receiver, and later on resume the receiving process on another device. It is possible that the receiver is activated on a terminal that is behind a firewall because a JMS subscriber initiates a connection to the JMS server, possibly over HTTP, and maintains it active while waiting for message publications from the server.

First, the publish/subscribe service should store the published messages matching user's subscription during disconnections. JMS offers the possibility to define durable subscriptions which remain active while the receiver is disconnected. The JMS server will store messages for a durable subscriber and deliver them when the durable subscriber reconnects.

The second problem is how to activate subscriptions on a JMS receiver. Note that the JMS receiver has no knowledge about subscriptions that are processed through the m-NewsBoard Web interface and activated by the publish/subscribe service. The publish/subscribe management component needs to notify a JMS receiver application about changes in user subscriptions. We have decided to implement the interaction between the JMS receiver and the publish/subscribe management component through

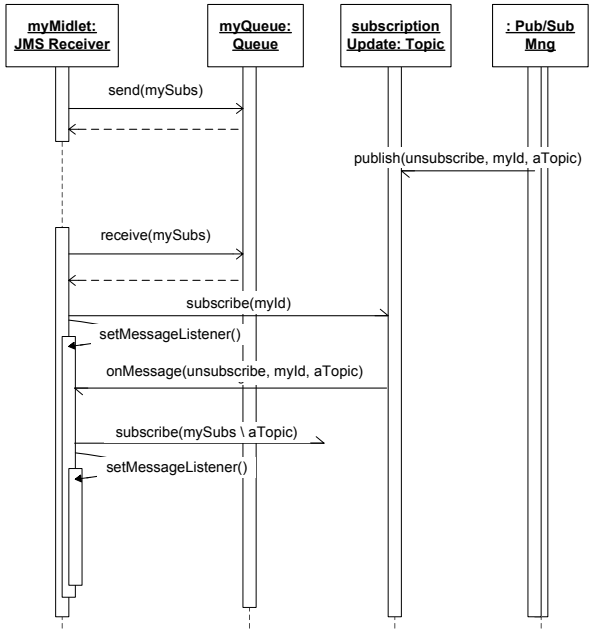

Figure 7.11: Sequence diagram for JMS-based delivery

queues and topics on the JMS server. During the JMS receiver bootstrap, the receiver subscribes to a special JMS topic - `subscriptionUpdate` - with a filter requesting that the user id of the subscription update message matches the subscriber id. Accordingly, the publish/subscribe service publishes the data about the changes in user subscriptions through the topic `subscriptionUpdate` and notifies the receiver that the user has changed the subscription. Consequently, in case of a subscription to a new topic, the receiver sends a new subscription request to the JMS server and starts a new message listener for the specified topic. In case of unsubscription an existing message listener is stopped.

The third problem is the storage and update of user subscriptions during disconnections since they cannot be stored on a terminal. User subscriptions are stored in a user's JMS queue: Prior to stopping the receiver, current subscriptions are put in the queue. Accordingly, during receiver bootstrap, the subscriptions are fetched from the queue and a new message listener is created for each topic subscription. In case there have been changes of subscription during disconnection, this notification will be received through the topic `subscriptionUpdate`.

Figure 7.11 shows a sequence diagram related to terminal-independent operation of the JMS receiver. Prior to JMS receiver disconnection from a JMS server, active subscriptions are stored in a special receiver queue, `myQueue`. The JMS receiver and the publish/subscribe management component interact through the topic `subscriptionUpdate`. When the subscriber unsubscribes from `aTopic` through the m-NewsBoard's Web interface, the message is stored on the server because of receiver's durable subscription to the topic `subscriptionUpdate`. When the user restarts the JMS receiver, possibly from a different terminal, the stored subscriptions are read from the queue `myQueue`. Next, it reissues the durable subscription to the topic `subscriptionUpdate`, and

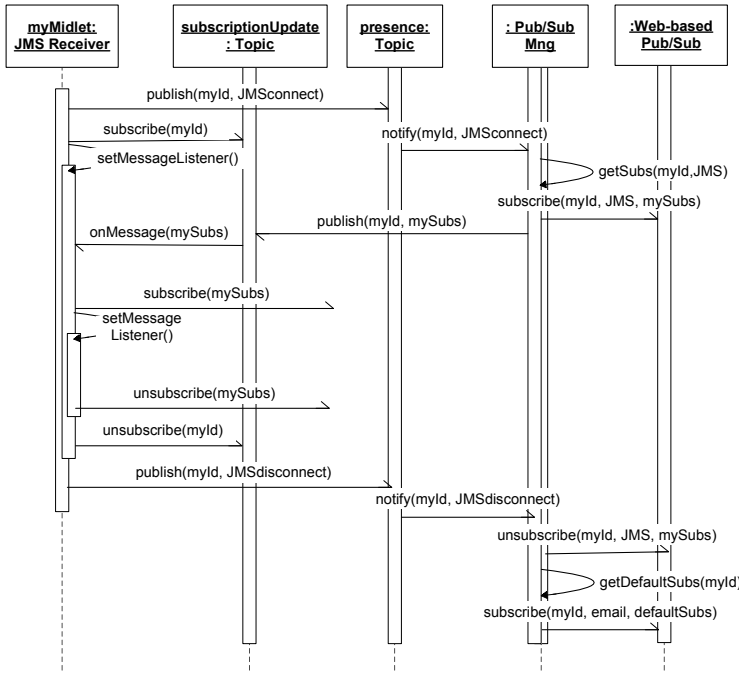

Figure 7.12: Sequence diagram for e-mail/JMS delivery

receives the previously published notification about user's unsubscription from the topic `aTopic`. Therefore, the receiver removes `aTopic` from the list of active subscriptions `mySubs` and reactivates active subscriptions.

Receiving the news using e-mail/JMS. The combination of e-mail and JMS notification delivery uses non-durable subscriptions during JMS-based notification delivery because notifications are sent in the form of e-mail messages during disconnections. Figure 7.12 depicts a sequence diagram showing the activation of JMS delivery using a JMS receiver, the operational JMS phase, a deactivation of the JMS receiver, and activation of the e-mail notification delivery. When a user activates a JMS receiver, the publish/subscribe management component is notified about the new presence status through the `presence` topic. The management component initiates user's JMS-based subscriptions through the publish/subscribe component, and subsequently notifies the JMS receiver to start listeners for the subscribed topics through the `subscriptionUpdate` topic. During the operational phase, in case the user changes his/her subscriptions, the receiver will be notified about subscription changes through the `subscriptionUpdate` topic. Before stopping the receiver, the information about disconnection is sent to the management component through the `presence` topic. The management component subsequently terminates JMS subscriptions and initiates e-mail subscriptions by sending a subscribe request to the publish/subscribe service.

7.4 Discussion

We have used m-NewsBoard as a case study to shown the applicability of the proposed reference architecture for the implementation of an example content dissemination service. m-NewsBoard is a personalized service that enables true user mobility: Users can define subscriptions and choose the preferred means for receiving the published content. The published news will be delivered to an active subscriber's receiver application independent of the applied device, or an e-mail message will be sent to the user's mail server in case the receiver application is inactive.

The presented implementation of the publish/subscribe service in the form of a Web service facilitates integration of publish/subscribe functionality into other applications. It offers a generalized interface which provides the common publish/subscribe constructs and maps general requests to method invocations that are specific to the transport component that performs the actual content dissemination. The main benefit of the proposed approach is the stability of the publish/subscribe generic interface and, at the same time, service flexibility and openness which enables the encapsulation of an arbitrary number of publish/subscribe-enabled components into the service. We are unaware of implemented solutions for generalized publish/subscribe Web-based services that are suitable for mobile scenarios. The existing publish/subscribe systems have specific proprietary APIs, although the offered functionality and communication patterns are similar. These systems are also primarily intended for use in stationary scenarios and offer limited support for mobility. Recently, a specification draft for Web Service Notification has been published [56] which addresses the problem of defining a topic-based publish/subscribe Web service. The draft is still in its initial phase, and the initial analysis shows that it has largely been influenced by JMS.

Based on the experience gained during the design and implementation of m-NewsBoard, we conclude that the prototype implementation of the Web-based publish/subscribe content dissemination service exhibits the following properties:

Flexibility and openness. The publish/subscribe service can employ different systems and protocols for content dissemination ranging from traditional mechanisms, such as mail and SMS, to recently developed infrastructures that implement the publish/subscribe interaction model. The generic interface of the publish/subscribe service assures system stability and simplifies the integration of publish/subscribe functionality into other value-added services. The addition of new transport components can improve service flexibility and performance, not its basic functionality: It will not affect other services requiring publish/subscribe.

Terminal independence and personal mobility. Terminal independence enables users to utilize publish/subscribe functionality from various devices in different networks. On top of terminal independence is personal mobility which offers a higher degree of mobility than terminal mobility and regards a user as an end communication point which is extremely important for the receipt of published notifications. Personal mobility does not depend on the publish/subscribe service itself, but needs to be provided by a particular transport component which performs the actual

delivery: In the presented JMS solution we have shown the ability to design a receiver using publish/subscribe principles that is able to receive the content in push-style regardless of the device where the receiver currently runs. Mail readers for various mobile devices, ranging from desktop computers, to mobile phones are available. The main drawback of mail in mobile scenarios is its pull-style operation which is inappropriate for networks with bandwidth limitations and intermittent connection. A better solution would employ SMS or MMS instead of mail in mobile scenarios.

Scalability. Service scalability depends heavily on the performance characteristics of the components that are used for transporting the content. Here we compare e-mail and JMS, the two components that are used in the prototype implementation. The solution that employs mail servers for content delivery is not scalable because of serious resource consumption if the number of receivers is high: Each mail message is duplicated and sent separately to each subscriber even if all subscribers use the same mail server. This puts a high processing load on the sender's mail server and causes high network bandwidth consumption. The existing JMS implementations differ in performance and scalability and load tests are needed to evaluate the performance of each solution for a particular application domain. The common JMS server implementations have a centralized architecture and may become a performance bottleneck in case of a large number of publishers and subscribers scattered in a wide area network. However, distributed JMS server solutions can significantly improve performance and scalability as discussed in Section 5.4.

Chapter 8

Conclusion

Even though notification services in mobile networks are gaining wide acceptance, they currently offer limited support for service customization and personalization. Service users can simply subscribe to a predefined topic, and receive all notifications published on the topic in the form of SMS or MMS messages. The concepts found in publish/subscribe systems offer means to remedy this limitation: Expressive content-based subscriptions enable users to describe the type of notifications that are of interest to them. The next limitation is that notifications are delivered to a single terminal regardless of the user's presence status, or the terminal the user is currently applying. Personal mobility management is therefore needed to extend the service and offer flexible usage scenarios to service users.

The thesis has presented a solution for a flexible, personalized content dissemination service that supports personal mobility of end users. The service serves as an information bus with filtering capabilities: Publishers can publish the content for numerous subscribers who define subscriptions to express their interest in receiving certain content types. The service compares each published notification to defined subscriptions and delivers the notification only to subscribers with a matching subscription. Notifications are delivered in push-style to the current subscriber's communication point in accordance with the subscriber's presence status.

8.1 Contributions

The thesis focuses on two aspects of content dissemination: the design and evaluation of a mobility-enabled publish/subscribe middleware, and the design of a software architecture for content dissemination services that uses the publish/subscribe middleware as its basic communication component.

Mobility-enabled publish/subscribe middleware. The thesis proposes an *event-based model* for distributed publish/subscribe systems supporting client mobility: The model defines the events that can occur in the system – *publish*, *subscribe*, *unsubscribe*, *notify*, *connect*, and *disconnect* – and change the system state. We introduce proxy subscribers and proxy publishers to model the distribution of system brokers: Proxy publishers represent publishers connected to neighboring brokers, while

proxy subscribers model subscribers residing on neighboring broker, and enable the communication and interaction between the brokers. We use the proposed model to define the routing algorithms for selective dissemination of notifications to subscribers that are mobile and potentially disconnected from a distributed system.

The defined *routing algorithms* use a novel principle that relies on notification persistency to solve the mobility problem in publish/subscribe systems: Notification publishers define the validity period for published notifications, the system stores notification during the validity period, and delivers valid notifications to subscribers when they reconnect to the publish/subscribe system. If a subscriber connects to the system after notification expiry, the notification will not be delivered to the subscriber. Two different routing algorithms that use persistent notifications have been defined: routing based on subscription equality, and routing based on subscription covering.

To validate the proposed model and the routing solution, we have implemented a prototype system that can be distinguished from other publish/subscribe implementations by the inherent support for publisher and subscriber mobility. Further on, we have used the prototype implementation to evaluate the performance of the proposed routing solution, and to compare it with the approach based on queues. The evaluation results show that the routing solution using persistent notifications is superior to the queuing approach with respect to broker memory consumption and scalability. It places less load on service brokers in case of the increased number of subscribers in the system which can be expected in real systems, and does not cause significant performance degradation in terms of notification delay when compared to the solution that uses queues. Best to the author's knowledge, this is the first evaluation of the publish/subscribe system performance in a mobile setting that provides performance measures regarding the broker load, notifications delay, and bandwidth consumption.

Content dissemination service architecture. We have designed an architecture for a mobile content dissemination service that enables the delivery of personalized and customized content to mobile users. The architecture is composed of the components that have been identified through the analysis of service usage scenarios. It uses a publish/subscribe middleware for realizing the interaction between service users, and a special personal mobility component for maintaining the user's presence information. We have designed a solution for a Web-based publish/subscribe service that offers a generic set of methods for the implementation of publish/subscribe interactions, and uses other components, such as mail, SMS, MMS, or JMS for the actual content transport. The design of the personal mobility component proposes the procedures for the update and retrieval of presence data, and analyzes the issues regarding security.

Finally, we have used m-NewsBoard, a news dissemination service, as a case study to evaluate the applicability of the proposed architecture. The m-NewsBoard system offers personalized news dissemination and enables true user mobility: Users can define subscriptions and choose the means for receiving the published content. We currently support e-mail and JMS-based delivery: The published news is delivered to an active JMS receiver independent of the applied device, or an e-mail message is sent to the user's mail server in case the receiver application is inactive. The m-NewsBoard system

utilizes a Web-based implementation of a publish/subscribe service that offers a generalized interface with the common publish/subscribe constructs, and maps general requests to method invocations that are JMS or e-mail specific. The main benefit of the approach is the stability of the publish/subscribe generic interface and, at the same time, service flexibility and openness which enables the encapsulation of an arbitrary number of publish/subscribe-enabled components into the service. A solution for terminal independence and personal mobility is implemented using the publish/subscribe and queue-based communication capabilities offered by the JMS.

8.2 Future Work

The thesis has given answers to certain questions related to mobile publish/subscribe systems, and content dissemination services. However, a number of interesting research problems have been identified for the future work.

The initial analysis shows that the presented publish/subscribe model and routing algorithms offer a solution for the design of a scalable distributed JMS-based broker network that can support client mobility. Service implementation and experimental evaluation are needed to investigate such claims. Further on, the presented model and algorithms can be extended by publisher advertisements: Evaluation studies show that the usage of advertisements in publish/subscribe systems reduces the traffic generated by control messages in distributed publish/subscribe systems [79]. We currently support routing algorithms that are based on reverse path forwarding: It would be interesting to investigate and evaluate the performance of other approaches, such as the core-based tree routing, and probabilistic gossip-based algorithms [64], in mobile settings. In the current publish/subscribe system design we assume that the system is fault-free: Mechanisms for designing a fault tolerant solution should be further investigated.

Moreover, further analysis of the recent attempts to publish/subscribe service standardization are needed. For example, we should investigate the compliance of the Web Service Notification initiative [56] with the proposed Web-based publish/subscribe service design. Another interesting extension of the content dissemination service is related to user geographical location. The integration of location status in the presence data can be used to offer location-based content dissemination.

Bibliography

[1] 3rd Generation Partnership Project. Virtual Home Environment/Open Service Access; (3GPP TS 23.127 V6.0.0), December 2002. http://www.3gpp.org.

[2] 3rd Generation Partnership Project. IP Multimedia Subsystem (IMS); Stage 2 (3GPP TS 23.228 V6.3.0), September 2003. http://www.3gpp.org.

[3] 3rd Generation Partnership Project. Presence Service; Architecture and functional description; Stage 2 (3GPP TS 23.141 V6.4.0), September 2003. http://www.3gpp.org.

[4] K. Aberer, A. Datta, and M. Hauswirth. Efficient, self-contained handling of identity in Peer-to-Peer systems, 2004. To be published in IEEE Transactions on Knowledge and Data Engineering (second quarter 2004).

[5] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 53–61. ACM Press, 1999.

[6] I. F. Akyildiz and J. S. M. Ho. On location management for personal communication networks. *IEEE Communications Magazine*, 34(9):138–45, September 1996.

[7] I. F. Akyildiz, J. McNair, J. S. M. Ho, H. Uzunalioğlu, and W. Wang. Mobility management in next-generation wireless systems. *Proceeding of the IEEE*, 87(8):1347–1384, August 1999.

[8] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[9] Apache XML Project. Xalan-Java version 2.5.2, 2004. http://xml.apache.org/xalan-j/.

[10] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, March 2000.

[11] B. Badrinath, A. Fox, L. Kleinrock, G. Popek, P. Reiher, and M. Satyanarayanan. A conceptual framework for network and client adaptation. *Mobile Networks and Applications*, 5(4):221–31, December 2000.

[12] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[13] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT): An architecture for scalable inter-domain multicast routing. In *Proceedings of ACM SIGCOMM'93*, pages 85–95. ACM Press, September 1993.

[14] S. Banerjee and B. Bhattacharjee. A comparative study of application layer multicast protocols, 2002. Submitted for review. http://citeseer.nj.nec.com/banerjee01comparative.html.

[15] K. Betz. A scalable stock web service. In *Proceedings of the 2000 International Conference on Parallel Processing, Workshop on Scalable Web Services*, pages 145–150, Toronto, Canada, 2000. IEEE Computer Society.

[16] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[17] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[18] E. A. Brewer, R. H. Katz, Y. Chawathe, S. D. Gribble, T. Hodes, G. Nguyen, M. Stemm, T. Henderson, E. Amir, H. Balakrishnan, A. Fox, V. N. Padmanabhan, and S. Seshan. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications*, 5(5):8–24, October 1998.

[19] G. Bricconi, E. Di Nitto, A. Fuggetta, and E. Tracanella. Analyzing the behavior of event dispatching systems through simulation. In *Proceedings of the 7th International Conference on High Performance Computing*, pages 131–140, December 2000.

[20] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 443–52, 2001. http://www-2.cs.cmu.edu/ ~chaki/publications/ICSE-2001.pdf.

[21] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, December 2003.

[22] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, 1998.

[23] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 219–227. ACM Press, July 2000.

[24] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[25] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicat infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, October 2002.

[26] M. Cherniack, M. J. Franklin, and S. Zdonik. Expressing user profiles for data recharging. *IEEE Personal Communications*, 8(4):32–8, August 2001.

[27] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[28] W. Consortium. Composite Capabilities/Preference Profiles, September 2003. http://www.w3.org/Mobile/CCPP/.

[29] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 24: Minimum Spanning Trees. MIT Press, 1990.

[30] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[31] G. Cugola and E. D. Nitto. Using a publish/subscribe middleware to support mobile computing. In *Proceedings of the Advanced Topic Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001*, November 2001.

[32] G. Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering*, pages 261–270, April 1998.

[33] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–50, September 2001.

[34] M. Day, J. Rosenberg, and H. Sugano. A Model for Presence and Instant Messaging, February 2000. RFC 2778. http://www.ietf.org/rfc/rfc2778.txt.

[35] S. E. Deering and D. R. Cheriton. Multicast routing in datagram networks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–111, 1990.

[36] A. Devlić and I. Podnar. Location-aware content delivery service using publish/subscribe. In *Proceedings of tcmc 2003*, March 2003.

[37] S. Dustdar, H. Gall, and M. Hauswirth. *Software-Architekturen für Verteilte Systeme*. Springer Verlag, 2003.

[38] S. Dustdar, H. Gall, and R. Schmidt. Web services for groupware in distributed and mobile collaboration. Technical Report TUV-1841-2003-24, Distributed Systems Group, Technical University of Vienna, 2003.

[39] A. El-Sayed, V. Roca, and L. Mathy. A survey of proposals for an alternative group communication service. *IEEE Network*, 17(1):2–7, 2003.

[40] W. Emmerich. Software engineering and middleware: A roadmap. In *The Future of Software Engineering - 22th International Conference on Software Engineering (ICSE 2000)*, pages 117–129. ACM Press, May 2000.

[41] P. T. Eugster, R. Boichat, R. Guerraoui, and J. Sventek. Effective multicast programming in large scale distributed systems. *Concurrency and Computation: Practice and Experience*, 13(6):421–447, May 2001.

[42] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[43] P. T. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001.

[44] P. T. Eugster, R. Guerraoui, and F. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 252–276. Springer-Verlag, June 2000.

[45] P. T. Eugster, R. Guerraoui, and J. Sventek. Type-based publish/subscribe. Technical report, Distributed Programming Laboratory, Swiss Federal Institute of Technology, June 2000.

[46] P. Felber, C.-Y. Chan, M. Garofalakis, and R. Rastogi. Scalable filtering of XML data for web services. *IEEE Internet Computing*, 7(1):49–57, January/February 2003.

[47] A. Festag, H. Karl, and G. Schäfer. Current developments and trends in handover design for All-IP wireless networks. Technical Report TKN-00-007, Telecommunication Networks Group, Technical University Berlin, Germany, 2000.

[48] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, volume 2672 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, June 2003.

[49] L. Fiege, G.Mühl, and F. C. Gärtner. Modular event-based systems. *Knowledge Engineering Review*, 17(4):359–388, 2003.

[50] G. Fox and S. Pallickara. JMS compliance in the Narada event brokering system. In *Proceedings of the International Conference on Internet Computing*, pages 391–402, 2002.

[51] M. J. Franklin and S. B. Zdonik. A framework for scalable dissemination-based systems. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pages 94–105, Atlanta, GA, USA, October 1997.

[52] N. Freed and N. Borenstein. Multipurpose Internet Mail Extension (MIME) Part Two: Media Types, November 1996. RFC 2046. http://www.ietf.org/rfc/rfc2046.txt.

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[54] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering, 2nd edition*. Prentice Hall, 2002.

[55] D. J. Goodman. The wireless Internet: Promises and challlenges. *IEEE Computer*, 33(7):36–41, July 2000.

[56] S. Graham and P. Niblett (editors). Web Services Notification, specification draft, January 2004. http://xml.coverpages.org/ws-notification200401.pdf.

[57] M. Hauswirth. *Internet-Scale Push Systems for Information Distribution—Architecture, Components, and Communication*. PhD thesis, Distributed Systems Group, Technical University of Vienna, October 1999.

[58] M. Hauswirth and M. Jazayeri. A component and communication model for push systems. In *Proceedings of the ESEC/FSE 99 – Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Fundations of Software Engineering*, September 1999.

[59] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE'01)*, pages 27–34, May 2001.

[60] J. F. Huber, D. Weiler, and H. Brand. UMTS, the mobile multimedia vision for IMT-2000: A focus on standardization. *IEEE Communications Magazine*, 38(9):129–136, September 2000.

[61] C. Huitema. *Routing in the Internet, 2nd ed.*, chapter 12: IP Multicast Routing. Prentice Hall, 2000.

[62] H.-A. Jacobsen. Middleware services for selective and location-based information dissemination in mobile wireless networks. In *Proceedings of the Advanced Topic Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001*, November 2001.

[63] S. Kapp. 802.11: Leaving the wire behind. *IEEE Internet Computing*, 6(1):82–85, January/February 2002.

[64] A. Kermarrec, L. Massoulie, and A. Ganesh. Reliable probabilistic communication in large-scale information dissemination systems, 2000.

[65] E. Kirda, C. Kerer, and M. Jazayeri. Supporting multidevice enabled web services: Challenges and open problems. In *Proceedings of the 10th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE Computer Society, June 2001.

[66] T. Kunz, A. A. Siddiqi, and J. Scourias. The peril of evaluating location management proposals through simulations. *Wireless Networks*, 7(6):635–643, 2001.

[67] G. Le Bodic. *Mobile Messaging Technologies and Services: SMS, EMS and MMS*. Wiley, 2003.

[68] S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial: Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993.

[69] T. Liao. Global information broadcast: An architecture for internet push channels. *IEEE Internet Computing*, 4(4):16–25, July/August 2000.

[70] Y.-B. Lin and I. Chlamtac. *Wireless and Mobile Network Architectures*. Wiley, 2001.

[71] I. Lovrek, M. Matijašević, and G. Ježić. Pokretljivost u mrežama. In A. Bažant et al., editor, *Osnovne arhitekture mreža*. Element, 2003.

[72] P. Maniatis, M. Roussopoulos, E. Swierk, K. Lai, G. Appenzeller, X. Zhao, and M. Baker. The Mobile People Architecture. *ACM Mobile Computing and Communications Review*, 3(3), July 1999.

[73] C. Mascolo, L. Capra, and W. Emmerich. Middleware for mobile computing (A Survey). In *Advanced Lectures on Networking - Networking 2002 Tutorials*, volume 2497 of *Lecture Notes in Computer Science*, pages 20–58. Springer-Verlag, May 2002.

[74] R. Meier. State of the art review of distributed event models. Technical Report TCD-CS-00-16, Dept. of Computer Science, Trinity College Dublin, Ireland, 2000.

[75] M. Mićin. *Usmjeravanje poruka u distribuiranim sustavima objavi/pretplati (In Croatian)—Routing messages in publish/subscribe systems*. Diploma Thesis. FER, University of Zagreb, Croatia, June 2003.

[76] R. Monson-Haefel and D. A. Chappell. *Java Message Service*. O'Reilly & Associates, 2001.

[77] G. Mühl. Generic constraints for content-based publish/subscribe systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS'01)*, volume 2172 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, 2001.

[78] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.

[79] G. Mühl, L.Fiege, F. C. Gärtner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proceedings of the 10th IEEE International Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, pages 167–176. IEEE Computer Society, October 2002.

[80] D. Muhoberac. *Implementacija usluge objavi-pretplati primjenom tehnologije Web Services (In Croatian)—Implementation of a publish/subscribe Web service*. Diploma Thesis. FER, University of Zagreb, Croatia, June 2004. To be published.

[81] C. Noble. System support for mobile, adaptive applications. *IEEE Personal Communications*, 2(3):44–49, February 2000.

[82] Object Management Group. CORBA event service specification, version 1.1, March 2001. http://www.omg.org/technology/documents/formal/event_service.htm.

[83] Object Management Group. CORBA notification service, version 1.0.1, August 2002. http://www.omg.org/technology/documents/formal/notification_service.htm%.

[84] ObjectWeb Open Source Middleware. JORAM - Java Open Reliable Asynchronous Messaging (release 3.6.0), August 2003. http://www.objectweb.org/joram/.

[85] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus—an architecture for extensible distributed systems. In B. Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, Asheville, NC, USA, December 1993. ACM Press.

[86] C. Peersman, S. Cvetkovic, P. Griffiths, and H. Spear. The global system for mobile communications short message service. *IEEE Personal Communications*, 6(3):15–23, June 2000.

[87] J. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/subscribe on the Web at extreme speed. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 627–630, Cairo, Egypt, September 2000. Morgan Kaufmann Publishers.

[88] C. E. Perkins. Mobile networking through Mobile IP. *IEEE Internet Computing*, 2(1):58–69, January-February 1998.

[89] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 22rd International Conference on Distributed Computing Systems - Workshops (ICDCS 2002 Workshops)*, pages 611–618. IEEE Computer Society, July 2002.

[90] P. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*. ACM Press, June 2003.

[91] E. Pitoura and G. Samaras. Locating objects in mobile computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571–592, July/August 2001.

[92] I. Podnar. Publish/subscribe middleware concepts. Technical Report FER-ZZT-2002-12-01, Department of Telecommunications, FER, University of Zagreb, December 2002.

[93] I. Podnar. Web-based mobile content dissemination service with publish/subscribe. Technical Report FER-ZZT-2003-09-01, Department of Telecommunications, FER, University of Zagreb, September 2003.

[94] I. Podnar, M. Hauswirth, and M. Jazayeri. Mobile Push: Delivering content to mobile users. In *Proceedings of the 22nd International Conference on Distributed Computing Systems - Workshops (ICDCS 2002 Workshops)*, pages 563–568. IEEE Computer Society, July 2002.

[95] I. Podnar and K. Pripužić. m-NewsBoard: A news dissemination service for mobile users. In *Proceedings of the 7th International Conference on Telecommunications (ConTEL 2003)*, pages 205–211. FER, University of Zagreb, June 2003.

[96] G. Pospischil, J. Stadler, and I. Miladinović. Location-based push architectures for the mobile internet. In S. Dixit and R. Prasad, editors, *Wireless IP and Building the Mobile Internet*, pages 503–524. Artech House, 2003.

[97] K. Pripužić. *Oblikovanje i razvoj aplikacije za isporuku višemedijskih poruka u mobilnom okruženju (In Croatian)—Design and implementation of an application for disseminating multimedia messages in mobile environments*. Diploma Thesis. FER, University of Zagreb, Croatia, September 2003.

[98] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the 3rd International Workshop on Networked Group Communication*, November 2001.

[99] P. Reinbold and O. Bonaventure. IP micro-mobility protocols. *IEEE Communications Surveys and Tutorials*, 5(1):40–57, Third Quarter 2003.

[100] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the 6th European conference on Foundations of Software Engineering (ESEC/FSE '97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 344–360. Springer / ACM Press, 1997.

[101] M. Roussopoulos, P. Maniatis, E. Swierk, K. Lai, G. Appenzeller, and M. Baker. Person-level routing in the Mobile People Architecture. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, October 1999.

[102] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware 2001*, pages 329–350, November 2001.

[103] A. K. Salkintzis. Wide-area wireless IP connectivity with the general packet radio service. In S. Dixit and R. Prasad, editors, *Wireless IP and Building the Mobile Internet*, pages 27–47. Artech House, 2003.

[104] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 1–7, May 1996.

[105] H. Schulzrinne and J. Rosenberg. The Session Initiation Protocol: Internet-centric signaling. *IEEE Communication Magazine*, 38(10):134–141, October 2000.

[106] H. Schulzrinne and E. Wedlund. Application-layer mobility using SIP. *Mobile Computing and Communications Review*, 4(3):47–57, July 2000.

[107] Softwired. iBus//Mobile developer's manual release 3.1., August 2002. http://www.softwired-inc.com.

[108] Sun Microsystems, Inc. Java Message Service Specification version 1.1, 2002. http://java.sun.com/products/jms/.

[109] Sun Microsystems, Inc. JavaSpaces Service Specification version 1.2.1, April 2002. http://wwws.sun.com/software/jini/specs/jini1.2html/js-title.html.

[110] Sun Microsystems, Inc. Java Remote Method Invocation (Java RMI), 2004. http://java.sun.com/products/jdk/rmi/index.jsp.

[111] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness–Transparent information delivery for mobile and invisible computing. In *Proceeding of the IEEE International Symposium on Cluster Computing and the Grid*, pages 277–285. IEEE Computer Society, May 2001.

[112] S. Tai and I. Rouvellou. Strategies for integrating messaging and distributed object transactions. In *Middleware 2000*, volume 1795 of *Lecture Notes in Computer Science*, pages 308–330. Springer-Verlag, 2000.

[113] A. S. Tanenbaum. *Computer Networks, 3rd edition*, chapter 7.4: Electronic mail. Prentice Hall, 1996.

[114] A. S. Tanenbaum. *Computer Networks, 3rd edition*, chapter 7.5: Usnet news. Prentice Hall, 1996.

[115] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*, chapter 12: Distributed coordination-based systems. Prentice Hall, 2002.

[116] TIBCO Software Inc. TIBCO Rendezvous (version 7.2), 2003. http://www.tibco.com/solutions/products/active_enterprise/rv/default.js%p.

[117] B. Turk. *Usporedba algoritama za usmjeravanje poruka u sustavima objavi-pretplati (In Croatian)—Comparison of the routing algorithms in publish/subscribe systems*. Diploma Thesis. FER, University of Zagreb, Croatia, September 2004. To be published.

[118] W3C. XML Path Language (XPath), version 1.0, November 1999. http://www.w3.org/TR/xpath.

[119] W3C. Web Services Description Language (WSDL), version 1.1, March 2001. http://www.w3.org/TR/wsdl.

[120] W3C. SOAP Version 1.2 Part 0: Primer0, June 2003. http://www.w3.org/TR/soap12-part0/.

[121] W3C. Web services architecture, August 2003. http://www.w3.org/TR/ws-arch/.

[122] H. J. Wang, B. Raman, C.-N. Chuah, R. Biswas, R. Gummadi, B. Hohlt, X. Hong, E. Kiciman, Z. Mao, J. S. Shih, L. Subraimanian, B. Y. Zhao, A. D. Joseph, and R. H. Katz. ICEBERG: An Internet core network architecture for integrated communications. *IEEE Personal Communications*, 7(4):10–19, August 2000.

[123] E. Yoneki and J. Bacon. Pronto: MobileGateway with publish-subscribe paradigm over wireless network. Technical Report UCAM-CL-TR-559, Computer Laboratory, University of Cambridge, 2003.

[124] S. Young, D. Spanjol, and V. K. Garg. Control of discrete event systems modeled with deterministic Büchi automata. In *Proceedings of 1992 American Control Conference*, pages 2809–2813, Chicago, IL, 1992.

[125] A. Zeidler and L. Fiege. Mobility support with REBECA. In *Proceedings of the 23rd International Conference on Distributed Computing Systems - Workshops (ICDCS 2003 Workshops)*, pages 354–360, May 2003.

[126] Y. Zhao and R. E. Strom. Exploiting event stream interpretation in publish-subscribe systems. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, pages 219–228. ACM Press, August 2001.

[127] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.

Summary

The dissertation presents an architecture and an implementation of efficient and personalized content dissemination service targeting mobile users. The service enables information publishers to publish the content for numerous users based on the publish/subscribe interaction style. Service personalization is achieved through subscriptions: Users define subscriptions to express their interest in receiving certain content types. The published content contains non-realtime data of variable bandwidth demands (short text messages, images or video clips) and the publishing time is usually randomly determined. Furthermore, the service enables personal mobility, i.e., a user can receive the content in various networks applying different terminals.

The thesis investigates two aspects of content dissemination. Firstly, a mathematical model of distributed publish/subscribe systems is presented, followed by the definition of routing algorithms that support publisher and subscriber mobility. Secondly, the thesis proposes a software architecture for content dissemination services based on a publish/subscribe Web service.

Keywords

content dissemination, publish/subscribe middleware, mobility, event-driven model, routing algorithm, software architecture

Kratki sadržaj

Disertacija predlaže arhitekturu i implementaciju usluge za učinkovitu i personaliziranu isporuku sadržaja pokretnim korisnicima. Usluga omogućuje objavljivanje sadržaja na načelu objavi-pretplati namijenjenog velikom broju korisnika. Usluga je personalizirana jer korisnici pretplatom izražavaju interes za primanje određene vrste sadržaja. Sadržaj čine podaci koji se ne prenose u stvarnom vremenu, varijabilnih su prometnih karakteristika (kratke tekst poruke, slike ili video isječci), a trenutak njihovog objavljivanja je slučajni događaj. Usluga treba omogućiti pokretljivost osobe, tj. mogućnost primanja sadržaja u raznovrsnim mrežama i na različitim terminalima.

Disertacija daje dva pogleda na uslugu za isporuku sadržaja. Najprije je predložen matematički model koji opisuje distribuirane sustave objavi-pretplati, te su definirani algoritmi umjeravanja poruka koji podržavaju pokretljivost korisnika sustava. Potom je predložena arhitektura usluge za isporuku sadržaja temeljena na komponenti objavi-pretplati koja je oblikovana primjenom tehnologije *Web service*.

Ključne riječi

isporuka sadržaja, međuoprema objavi-pretplati, pokretljivost, model voden događajima, algoritam usmjeravanja, arhitektura programskog proizvoda

Curriculum Vitae

I was born on October 29th, 1973 in Zagreb. After finishing high school, natural science track, in Zagreb, I started the undergraduate program at the Faculty of Electrical Engineering and Computing, University of Zagreb, in 1992. I received my B.S. (Dipl.-Ing.) and M.S. degrees in electrical engineering with a major in telecommunications and information science from the University of Zagreb, in 1996 and 1999, respectively. The research topic of my Master's thesis was "Software Maintenance Process Analysis". I am currently a teaching assistant at the Faculty of Electrical Engineering, University of Zagreb. I have been affiliated with the Department of Telecommunications at the named Faculty since 1997. In 2000 and 2001 I was on leave from the University of Zagreb, working toward my Ph.D. as a research associate at the Information Systems Institute of the Technical University of Vienna, Austria. My current research interests include distributed information systems, publish/subscribe systems in particular, and services in mobile networks. I have published 13 papers on international conferences in the area of distributed systems and software maintenance. I am fluent in English, German, and Italian. I am a member of IEEE.

Životopis

Rođena sam 29. listopada 1973. u Zagrebu. Po završetku srednje škole (XV Gimnazija u Zagrebu), 1992. godine upisujem studij na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu, gdje sam i diplomirala u prosincu 1996. godine s temom "Optimalno pridjeljivanje valnih duljina u sveoptičkim mrežama s valnim multipleksom". Magistrirala sam u prosincu 1999. godine na Fakultetu elektrotehnike i računarstva, i time stekla znanstveni stupanj magistra znanosti iz područja tehničkih znanosti, polje Elektrotehnika, smjer Telekomunikacije i informatika. Tema mog magistarskog rada je "Analiza procesa održavanja programske opreme". Od veljače 1997. godine sam zaposlena na Zavodu za telekomunikacije Fakulteta elektrotehnike i računarstva u zvanju asistenta. Tijekom 2000. i 2001. godine sam radila kao znanstveni suradnik na Institutu za informacijske sustave Tehničkog sveučilišta u Beču radi znastvenog usavršavanja u okviru doktorskog studija. Moja područja istraživanja su distibuirani informacijski sustavi s naglaskom na sustave *objavi-pretplati* i usluge u pokretnim mrežama. Objavila sam 13 radova na međunarodnim konferencijama iz područja distribuiranih sustava i održavanja programske podrške. Govorim engleski, njemački i talijanski jezik. Član sam udruženja IEEE.