Supporting Mobility in Content-Based Publish/Subscribe Middleware

Ludger Fiege[1], Felix C. Gärtner[2], Oliver Kasten[3], and Andreas Zeidler[1]

[1] Darmstadt University of Technology (TUD), Department of Computer Science, Databases and Distributed System Group, 64283 Darmstadt, Germany, {fiege,az}@dvs1.informatik.tu-darmstadt.de
[2] Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Distributed Programming Laboratory, CH-1015 Lausanne, Switzerland, fgaertner@lpdmail.epfl.ch
[3] Swiss Federal Institute of Technology (ETHZ), Department of Computer Science, Distributed Systems Group, CH-8092 Zurich, Switzerland, oliver.kasten@inf.ethz.ch

Abstract. Publish/subscribe (pub/sub) is considered a valuable middleware architecture that proliferates loose coupling and leverages reconfigurability and evolution. Up to now, existing pub/sub middleware was optimized for static systems where users as well as the underlying system structure was rather fixed. We study the question whether existing pub/sub middleware can be extended to support *mobile and location-dependent applications*. We first analyze the requirements of such applications and distinguish two orthogonal forms of mobility: the system-centric physical mobility and an application-centric logical mobility (where users are aware that they are changing location). For logical-mobility we introduce *location-dependent subscriptions* as a suitable means to exploit the power of the event-based paradigm in mobile applications. Briefly spoken, a location-dependent subscription offers to express interest in all events which are related to a user's current location. We present efficient implementations for both forms of mobility within the content-based pub/sub middleware Rebeca. Our solutions draw much of their efficiency from the refined routing capabilities (namely, covering and merging) of the Rebeca system.

Keywords: Publish/subscribe, Mobility, Location-awareness

1 Introduction

Location-based services. The emergence of mobile computing has opened up a whole new field of services which can be offered for the benefit of the mobile user. Many such services can exploit the fact that the mobile device is aware of its current location. For example, car navigation systems use knowledge about current and past locations to aid drivers find their way through unknown cities. Location information can even be combined with other sources of data, e.g., the

weather report, information on traffic jams or free parking spaces. In such cases, the system can propose routes which avoid places where traffic is high or weather conditions are unpleasant, or can direct the driver to the nearest free parking space. All these are examples for what are called *location-based services*.

Publish/subscribe systems. A convenient way to construct location-based services is to build them using event infrastructures, such as those provided by *publish/subscribe systems* (pub/sub systems). The pub/sub system allows producers and consumers to exchange information based on message content rather than a particular destination identifier or address. This *loose coupling* of producers and consumers is the premier advantage of pub/sub systems in general. In this paper we study how to exploit these advantages in the context of mobile services.

Unfortunately, up to now research in pub/sub systems has mainly focused on *static* systems, i.e., systems where the clients (i.e., producers and consumers) do not move and the pub/sub infrastructure remains somewhat stable throughout the system's lifetime. A number of middleware systems have been developed and optimized for these system environments, e.g., Elvin [23], Gryphon [14], REBECA [10], and Siena [4]. If present at all, support for mobility is a concern of the application layer: some application detects the need to change a subscription either because the interest has changed or some external trigger becomes true.

Related work. Middleware for mobile computing has been the focus of much interest for some years (see [2] for a survey) but usually work has concentrated on classical synchronous middleware like CORBA (see for example [17]). Only recently, position papers have stated that pub/sub systems have an enormous potential to better accommodate the needs of large mobile communities [15,5].

In contrast to this paper, Huang and Garcia-Molina [13,12] approach the problem of supporting mobility in pub/sub systems by describing algorithms for a "new" middleware system tailored and optimized to mobile and ad-hoc networks. We are only aware of two papers [5,24] that try to extend existing pub/sub architectures. Cugola and Di Nitto [5] describe how to dynamically adapt the internals of the Jedi event system but their implementation model is based on multicast communication. Similarly, Sutton, Arkins and Segall [24] present an extension to the Elvin system for supporting mobility. The main idea is a central proxy which handles subscriptions on behalf of the client. Obviously, this proxy can become a bottleneck and their solution does not scale well. In contrast to that, our solution is fully distributed and therefore more scalable. Moreover, none of the referenced papers addresses the fundamental question on how to integrate location-awareness into the pub/sub paradigm.

Supporting mobility in pub/sub middleware. In this paper, we argue that support for mobility should be an issue of the pub/sub middleware itself and not be delegated to the application layer. But instead of designing a new type of "mobile" middleware from scratch, we devise a feasible deployment path which takes the existing and highly optimized systems as a basis and modifies their internals in

such a way that they remain suitable for the static applications which they were originally designed for. This approach has several obvious advantages. Firstly, existing event-based applications relying on pub/sub middleware must not be changed and can run simultaneously with new mobile applications using the same middleware. And secondly, static applications still exploit the high degree of optimization built-in to the existing systems. A third advantage, which might not be so obvious but which we substantiate in this paper, is that mechanisms for mobile applications can be efficiently embedded into currently available middleware systems, i.e., they can make use of the very means that make these systems so efficient in the static case.

We develop mobility support in the context of the pub/sub middleware REBECA[8], a system that supports *content-based filtering*. Briefly spoken, content-based filtering allows to define a subscription as an expression over the *entire content* of a notification.

We provide support for two different and orthogonal types of mobility. The first type of mobility is called *physical* mobility. In physical mobility, clients may disconnect from the pub/sub system (due to power-saving requirements or the network characteristics) and later reconnect to the system, possibly at a different place. Physical mobility is similar to the concept of *roaming* or *terminal mobility* known from other types of mobile networks and is a system-oriented type of mobility. This means that applications are not necessarily aware of the fact that the client is moving, allowing existing applications to be transfered to mobile environments.

The second type of mobility which we support is called *logical mobility*. In logical mobility, the client remains attached to the same broker all the time but has a local awareness of changing locations. The concept of logical mobility is closely tied to the novel concept of *location-dependent subscriptions*, which is introduced in this paper. As an example, consider a car looking for a free parking space in the street it is currently driving along. In this situation it may subscribe to "all free parking spaces on Rebeca Drive". A car pulling out a couple of blocks down the road may generate a matching notification which is then delivered to the subscriber who will be glad to find a place to park. However, if Rebeca Drive is a very long street, the same driver will also receive notifications about free parking spaces very far down the road (or behind him) which are impossible to reach in good time. What the user would like to do is to specify a subscription such that he receives all notifications about "vacancies in the vicinity of his current location". We call these subscriptions *location-dependent*.

Briefly spoken, we require that the client does not miss any interesting notification when being physically or logically mobile. This requirement imposes different restrictions on the solutions which we propose because logical mobility is orthogonal to physical mobility. Logical mobility deals not with system structure but rather system semantics. For logical mobility, the user (or application developer) must define a notion of (change of) location which is not necessary in physical mobility. In physical mobility the system must provide some form of buffering of notifications because of temporary disconnections. This paper

shows that both forms of mobility can be implemented efficiently in existing content-based pub/sub systems by exploiting the efficiency of refined content-based routing schemes.

Contributions. We summarize the contributions of this paper as follows:

- We identify and separate different types of mobility in the context of content-based pub/sub systems and their concerns (physical vs. logical mobility).
- In the context of logical mobility, we introduce the concept of a *location-dependent subscription*.
- For physical mobility and logical mobility (i.e., location-dependent subscriptions) we present efficient solutions which are based on extensions of the existing pub/sub middleware REBECA.

Outline. This paper is structured as follows: We provide some basic background and terminology on content-based pub/sub and the REBECA system in Section 2. We then discuss in more detail the issues involved when supporting mobility using existing content-based pub/sub middleware in Section 3. Within the context of REBECA, we present a solution for physical mobility in Section 4 and a solution for logical mobility in Section 5. We conclude in Section 6.

2 Content-Based Publish/Subscribe

The pub/sub communication paradigm is increasingly used in many applications and areas of computer science[11,6,3]. The basic model is presented in the next paragraph, and out of the variety of existing flavors we select content-based, distributed pub/sub for our scenario in Section 2.2. The discussion is based on the REBECA notification service [8,9], which we use as basis for the proposed mobility support.

2.1 Publish/Subscribe Systems

A pub/sub system consists of a notification service and application processes, which are clients of the service. It allows the processes to exchange information based on message content rather than particular destination addresses. Processes communicate by generating and receiving asynchronous notifications [10]; a process can act both as producer and as consumer of notifications. A *notification* is a message that is not directed to a specific receiver, but published into the underlying communication infrastructure. This infrastructure is provided by a notification service whose responsibility is to convey messages from producers to consumers. Consumers issue subscriptions describing those notifications they are interested in, and the notification service has to ensure that only matching notifications are delivered. In some systems producers are required to issue advertisements that describe the notifications they are about to publish; they are used to optimize notification routing in the underlying infrastructure.

The expressiveness of the notification service is determined by its language used for specifying subscriptions and the data model of the transmitted notifications. In nearly all cases subscriptions act as filters on the notifications and three filtering schemes can be distinguished. Subject- and type-based addressing schemes use path expressions and type hierarchies to denote and select otherwise opaque notifications [22,1,7]. The most flexible scheme, however, is offered by content-based filtering that utilizes predicates on the entire content of a notification [18]. Together with the typically used name/value pairs data model, subscriptions look like: $(\textit{service} = \text{“parking”}), (\textit{location} = \text{“100 Rebeca Drive”}), (\textit{cost} < \text{“3 EURO”}), (\textit{car-type} \geq \text{“compact”})$. This approach is very popular in both research and industry because of its simplicity and extensibility [20]; the Java Message Service (JMS) uses a combination of subjects and content-based addressing.

The loose coupling of producers and consumers is the prime advantage of pub/sub systems. Producers are relieved from managing interested consumers and vice versa. The functionality of both can be concentrated on their key features, emphasizing the importance of the notification service for the system's overall functionality. The communication interface to the service is rather simple and consists of *pub*, *sub*, *unsub*, and *notify* calls only; the last one is an output function being called on the registered client to deliver a notification.

2.2 Content-Based Routing

The next paragraphs describe the system model of a distributed pub/sub system. The system is modeled as a distributed set of concurrent processes that communicate via asynchronous message passing over point-to-point, error-free[1] communication links. This model simplifies the presentation and alleviates the problems of using network level multicast, which still remains an implementation-dependent option to improve communication performance. Furthermore, messages are required to be delivered in FIFO order with respect to the sender. The individual processes are assumed to have local real-time clocks which are synchronized using a standard protocol like NTP. While we postulate that there is no upper bound on the message delivery delay, we assume that the delays satisfy some probability distribution so that an expected delivery time can be computed statistically.

The communication topology of the pub/sub system is given by a graph, which is assumed to be acyclic and connected (Fig. 1). The graph consists of brokers and clients (of the pub/sub service). Brokers are processes that route the notifications along multiple hops to the appropriate clients, which are application processes and the producers and consumers of notifications. Brokers use *messages* to forward notifications and administrative data. Three types of brokers are distinguished:

– Local brokers are part of the communication library loaded into the clients; they are not represented in the graph, but only used for implementation issues.

[1] Reliability issues are discussed elsewhere [19].

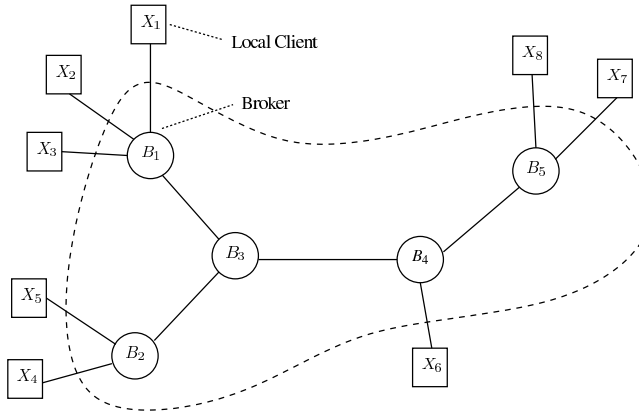

Fig. 1. Implementation of the distributed content-based routing framework of REBECA.

- Border brokers form the boundary of the distributed communication middleware and maintain connections to local brokers, i.e., the clients.
- Inner brokers are connected to other inner or border brokers and do not maintain any connections to clients.

Each broker maintains a routing table that determines the decision in which directions a notification is forwarded. A routing table in the content-based routing framework of REBECA [18] contains a list of link/filter pairs that describe the notifications for which subscribers are registered in the subgraph reachable via the given link. Every incoming notification is tested against the routing table's entries to determine the set of links with matching filters. In a second step the notification is forwarded to the respective next broker along these links. We assume the routing decision to be an atomic operation so that the end-to-end sender FIFO characteristic holds.

Besides exchanging notifications, brokers send control messages to maintain up-to-date routing tables. If a consumer C_i issues a subscription that contains a filter F, its border broker sets up a routing table entry (F, L_{C_i}) describing that a filter F is accepted at the link to which C_i is connected. The broker then forwards the information about a new subscriber to its neighbor brokers if necessary. The exact strategy is determined by a distributed content-based routing algorithm that defines how filtering of notifications at intermediary brokers is performed [19].

If filters are not compared to each other and routing tables just collect all active filters assigned to a given link without differentiating them in any way, the routing strategy is called *simple routing*. The obvious disadvantage is the growth of table size since a hundred different consumers with equal subscriptions would lead to hundred entries in each routing table.

A first improvement is to check and combine two filters if they are equal. More general, the *covering* routing strategy [4] tests whether a filter F_1 accepts

a superset of notifications of a second filter F_2, and in this case replaces all occurrences of F_2 assigned to the same link in the routing table. In a second step, if no cover can be found in a given set of filters, *merging* can be used to create new filters that are covers of existing ones [18]. Only the resulting merged filter is forwarded to neighbor brokers, where it covers and replaces the base filters.

3 Publish/Subscribe Systems and Mobility

In this section we analyze and discuss the basic issues involved when adding mobility support to a pub/sub infrastructure. We identify and define two orthogonal forms of mobility (physical and logical mobility) and discuss the requirements of a system supporting both types of mobility.

3.1 Mobility Issues in Publish/Subscribe Middleware

Mobile clients have many characteristics, among them the need to disconnect from the network for different reasons. Be it for geographical, administrative, or power saving reasons, being connected to the same broker all the time is no longer possible. Hence, we have to take into account that clients will disconnect from their border broker once in a while. The middleware has to deal with moving clients and the possibility that a disconnected client reconnects at the same or a different broker later.

A first step towards mobility is to enhance existing pub/sub middleware to allow for roaming clients so that existing applications can be used in mobile environments. This means that the interfaces for accessing the middleware and the applications on top are not required to change. More importantly, the quality of service offered by the middleware must not degrade substantially. The resulting location transparency is necessary to make existing applications mobile; e.g., stock quote monitoring seamlessly transfered from PCs to PDAs.

On the other hand, future applications do not want complete transparency, but rely on awareness of mobility. More specifically, mobility support should blend out unwanted phenomena, like disconnectedness, and enforce wanted behavior, like the location-awareness in location-based services. Consequently, extending the interface of the pub/sub middleware to facilitate location-awareness is a promising open issue, since most existing work concentrated on the transparency only.

When roaming, clients change (at least some portion of) the context they are operating in, and they might want to react to these changes; e.g., to adapt their subscriptions. However, an appropriate infrastructure support has to relieve the application from having to react "manually" to all changes. The middleware should rather offer an automated adaptation to context changes, i.e., facilitating location dependency. This leads to a different notion of mobility and we distinguish:

– *Physical mobility*: A client that is physically mobile disconnects for certain periods of time and have different border brokers along his itinerary through the infrastructure. Usually, a physically mobile client is not aware of its location changes (transparency).
– *Logical mobility*: A client that is logically mobile is aware of his location changes.

Physical and logical mobility are two orthogonal aspects of mobility. Since the physical layout of a pub/sub system is usually fixed and its layout does usually not correspond to geographical realities, it seems reasonable to separate the two notions of mobility. In this paper, we assume logical mobility to be a refinement of physical mobility in that a client remains connected to the same broker when roaming logically. The two notions have different quality of service requirements and therefore different solutions are developed to match both.

3.2 Physical Mobility

Physical mobility is similar to what in the area of mobile computing is called *terminal mobility* or *roaming*. A client accesses the system through a certain number of *access points* (GSM base stations, WaveLan access points, or border brokers). When physically moving, the client may get out of reach of one access point and move into the reach of a second access point. To maintain a connection and to hide phases of disconnectedness, the responsibility of providing access to the system is usually handed over from one access point to the other (involving some handover protocol).

Let us consider the quality of service requirements of a roaming client changing brokers. In order to make physical mobility transparent to the application the quality of service that a roaming client experiences should be the same as in non-mobile scenarios (cf. Sect. 2). Non-mobile broker networks are able to guarantee per sender FIFO ordering of notifications, e.g., if inter-broker communication is based on TCP/IP. Any handover protocol has to meet this requirement for roaming pub/sub clients.

Possible Solutions: One solution would be to rely on Mobile IP [16] for connecting clients to border brokers, hiding physically mobility in the network layer. The drawback, however, is that the communication is also hidden from the pub-/sub middleware, which is then not able to draw from any notification delivery localities or routing optimizations. Such an approach might only be feasible if the physical and logical layout of a given system is completely orthogonal.

A different, naïve solution to implement physical mobility would be to use sequences of sub-unsub-sub calls, simply registering a client at a new broker. When a client moves from border broker B_1 to B_2, it simply unsubscribes at B_1 and (re-)subscribes at B_2, without any support in the middleware. But a client may not detect leaving the range of a broker and is in this case not able to unsubscribe at its old location. Even more severely, during its time of disconnectedness, the client might miss several notifications or get duplicates, even if

notifications are flooded in the network and the location change is instantaneous. This problem is depicted in Figure 2.

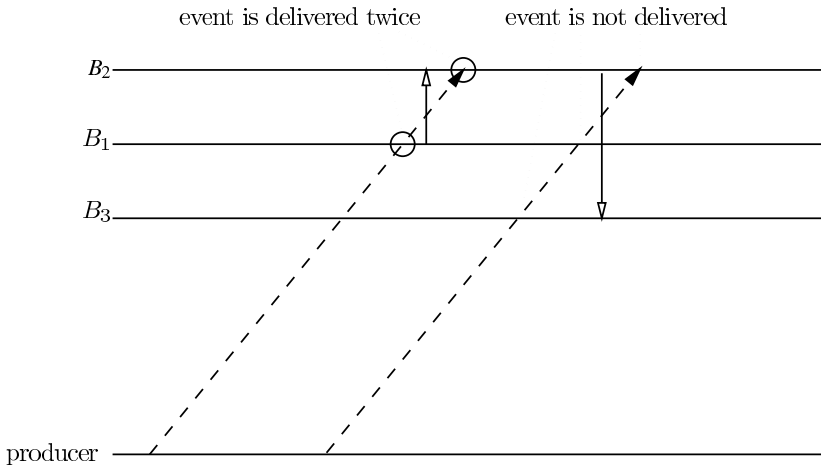

Fig. 2. Missing notifications in a flooding scenario.

3.3 Logical Mobility

While physical mobility is a rather technical issue invisible to the application, logical mobility involves location awareness. An example for logical mobility is when clients move around a house or building which is served by only one border broker. In this case, the user might be interested to receive just those notifications which refer to the room he is currently located in. Note that a client can be both logically and physically mobile at the same time.

A logically mobile client moving from one location to another, e.g., from one room to the other in a company building, will expect a frictionless change of location explicitly without a notable setup time after having changed from the own office to the conference room next door. The adaptation of some location dependent subscription should take place "instantaneously". Intuitively, we would like to experience the notion of being subscribed to "everything, everywhere, all the time" and increase the reactivity of the system to clients moving.

Location-Dependent Filters. A pub/sub system which offers location-dependent filters has the same interface as a regular pub/sub system (i.e., it offers the *pub*, *sub*, *notify* primitives). However, in specifying subscription filters for name/-value pairs referring to "*location*" it supports a new primitive to specify things like "all notifications where the attribute *location* equals my current location". More precisely, we postulate a specific marker *myloc* which can be used in a

subscription. The marker stands for a specific set of locations which depend on the current location of the client. For example, a client could issue a subscription for all free parking spaces in the vicinity of his current location as follows: $(service = \text{“parking”}), (location \in myloc), (car\text{-}type \geq \text{“compact”})$.

The set of locations associated with the marker is taken from a particular range L of locations. This set is application dependent and can, for instance, contain all the different rooms of a building, all the streets of a town, or all the geographical coordinates given by a GPS system up to a certain granularity. Given a notification with the attribute $(location \in x)$, the subscription $(location \in myloc)$ will evaluate to true for a particular client at location y if and only if $x \in myloc(y)$ where $myloc(y)$ is the specific set of locations associated with y. In this case we say that the notification matches the location-dependent filter.

The simplest form of $myloc(y)$ is simply the set $\{y\}$. In this case a notification matches the subscription if $x = y$. But in the car example, the car driver looking for a parking space might want to specify ($location =$ “at most two blocks away from *myloc*
In this case, *myloc* corresponds to all elements of L that satisfy this requirement.

A tentative but incomplete solution for logical mobility. While location-dependent filters are not directly supported by current pub/sub middleware, one might argue that it is not very difficult to emulate them on top of currently available systems in this case. The idea would be to build a wrapper around an existing system which follows the location changes of the users and transparently unsubscribes to the old location and subscribes to the new one when the user moves. However, depending on the internal routing strategy of the event system, it may lead to unexpected results. We have found that the routing strategies deployed in many existing content-based event systems such as Siena [4], Elvin [23], and Rebeca [8] lead to blackout periods where no notifications are delivered. The problem is that it usually takes an unnegligible time delay to process a new subscription. After subscribing to a filter, it takes some time t_d until the subscription is propagated to a potential source. Then it takes at least another t_d time until a notification reaches the subscriber. This phenomenon is depicted in Figure 3a. (Note that the delay t_d may be different for different notification sources and may change over time.) If the client remains at any new location less than $2t_d$ time, then the subscriber will “starve”, i.e, will receive little or no notifications.

An intuitive but inefficient solution. Another basic solution which can be immediately built using existing technology is again based on flooding. The local broker can then decide to deliver a notification to a client depending on the client's current location (see Figure 3b). Obviously, flooding prevents the blackout periods which were present in the previous solution, but it should be equally clear that flooding is a very expensive routing strategy especially for large pub/sub systems [21].

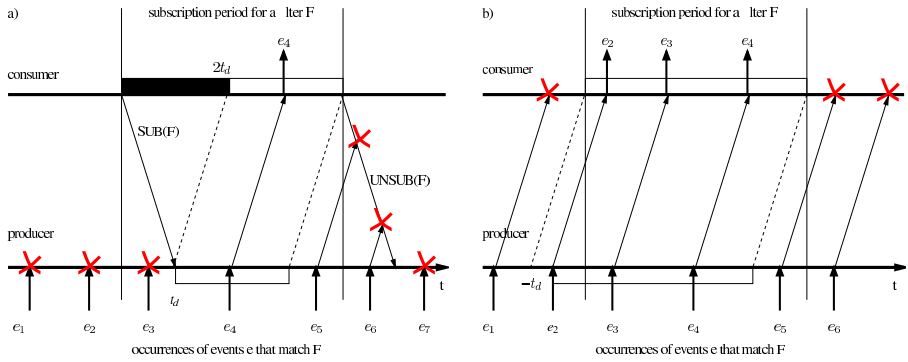

Fig. 3. Blackout period after subscribing with simple routing a) and flooding with client-side filtering b).

Quality-of-Service of logical mobility. Interestingly, while flooding is very expensive and therefore not desirable, it comes very close to the quality-of-service which we would like to achieve for logical mobility, namely to the notion of being subscribed to "everything, everywhere, all the time". The problem is that it is hard to precisely define the behavior of flooding without reverting to some unpleasantly theoretical constructions of operational semantics.

With logical mobility there is however no danger of receiving a notification twice because the consumer remains attached to the same "delivery path". The quality of service we require for logical mobility therefore is simply stated as follows: On change of location from x to y, all notifications should be delivered to the consumer "as if" flooding were used as underlying routing strategy. This statement is made a little more concrete in Figure 4 where the sequence of notifications generated by any consumer is divided into epochs which correspond to when the notification actually arrives at the consumer (the epoch borders between location y and z are drawn as a virtual notification $n_{y \to z}$). We require that all notifications matching the current location-dependent subscription from every such epoch must be delivered. Intuitively, the epochs define the semantics of flooding.

4 Notification Delivery with Roaming Clients

In this section we introduce an algorithm for extending standard REBECA brokers to cope with roaming mobile clients, maintaining their subscriptions as well as guaranteeing the required quality-of-service which was described in the previous section.

The solution sketched in this section is based in two assumptions: Firstly, we assume that brokers are able to install and maintain a buffer (or cache) for all notifications that pass through them for a certain period of time. This assumption is necessary to deal with temporary disconnects. If the disconnection

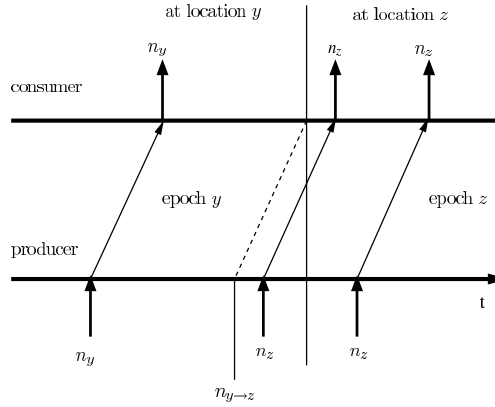

Fig. 4. Defining the quality-of-service for logical mobility using virtual notifications $n_{y\to z}$ that arrives at the consumer just at the time of the location change from y to z.

time is longer than the cache is able to store notifications, then it is impossible to guarantee uninterrupted notification delivery. Secondly, we assume that the underlying routing infrastructure uses advertisements. This is necessary to efficiently reroute notifications to a moving client in a distributed fashion. Both assumptions (cache and advertisements) are catered for in REBECA.

Apart from guaranteeing uninterrupted notification delivery, our algorithm also guarantees that the "old" border broker (i.e., the broker to which the roaming client was formaly attached) will eventually receive an equivalent to an explicit *sign-off* from the client, so that it can garbage collect all resources allocated to a specific client. In this process the algorithm also guarantees that any routing path to the old location related to a client will be deleted.

4.1 Example

Instead of directly diving into the details of the algorithm, we explain it with the help of an increasingly complex example. The example consists of two parts. In the first part we consider a very simple scenario with one mobile client and a single producer. The second part shows how the algorithm works with multiple producers.

Single producer example. We illustrate the relocation process using the simple example scenario on the left of Fig. 5. Assume client C is moving from the location with broker B_6 to another location with broker B_1. This refers to step 1 in the figure. After the client has detected the change of location and broker it re-issues a subscription together with the last received sequence number for this subscription (e.g., $(C, F, 123)$, with 123 as last known sequence number; step 3). Broker B_1 will detect that this client has moved and must be relocated. Note

that neither client nor broker need to have any knowledge about the old location B_6. Broker B_1 then starts the relocation process by sending a special message to its neigboring brokers.

The goal of the relocation process is to divert the old delivery paths to C from its old location to the new location. During this process, the brokers propagate the subscription through B_2 and B_3 to broker B_4. Here the old and new path from producer P to client C meet (dotted and dashed line, respectively). Broker B_4 is aware of this by inspecting the own routing table, its list of received advertisements, and comparing it to the subscription received. As B_4 has an old entry for this subscription, B_4 sends a fetch request $(C, F, 123, B4)$ along the old path to B_6 and already starts routing all newly received notifications from P along the *new* path.

When receiving the fetch request along the path to the old broker B_6, all broker along this path update their routing tables such that they are pointing into the direction of B_4. B_6 as last recipient replays all buffered events for (C, F) beginning with sequence number initially given by C to B_1 (here 124) and sends a message with a replay of all notifications received in the meantime along the path into the direction of B_4. As all intermediate brokers have already updated their routing tables, the replay eventually reaches B_1 via B_4 and is delivered to C. In the meantime, B_1 has buffered all notifications which have arrived for C and delivers the old messages from B_6 first before delivering the "new" messages from its own buffer to guarantee the correct delivery order.

Broker B_6 at the old location can garbage collect all resources formerly associated with C, so can B_5, resulting in the new routing path between P and C as shown in Fig.5.

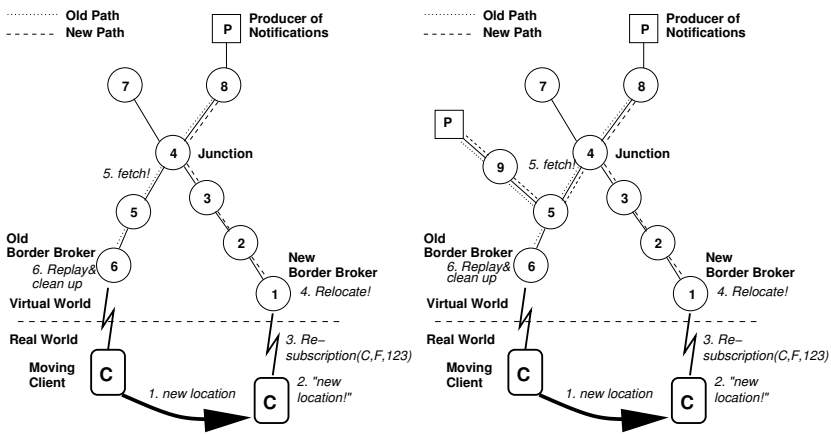

Fig. 5. A moving client scenario with one producer (left fig.) and more than one producer (right fig.).

Multiple producer example. In case there are multiple producers, reestablishing the new delivery paths towards the new broker of C is a little more complicated. Let us assume a scenario like the one depicted on the right side of Fig. 5. The scenario is the same like before except that client C was fed by more than one producer.

The only change in behavior of the processes is that every broker on the old path towards B_6 starting with B_4 has to check whether there is more than one advertisement for a producer matching filter F. If so, that broker is at a junction, like B_4, and hence the path towards B_4 must not be discarded. Only the part after passing the last junction can be deleted safely.

In order to determine which is the last broker at a junction the algorithm uses the fetch request in direction of the producers and the replay message in direction of the old broker to which the client was subscribed. On the way towards B_6 every broker with a junction towards a new producer replaces the broker identifier in the fetch request with its own. In the example, B_4 starts the process by sending a replay request to B_5. Since B_5 is the next junction, B_5 sends a replay request to B_6 with the broker identifier set to B_5 because no more junctions are on the path. The replay message on the other hand is used to delete the not needed final part of the path towards B_1. As soon as B_5 receives a message from B_6 and the identifier of B_5 equals the identifier contained in the message, B_5 "knows" it is the last broker with an additional producer on the path towards B_6 and therefore it cannot delete the routing table entry for (C, F) but has to modify it to point to B_4.

Compared to the first example the difference is obvious: While in the first scenario B_6 and B_5 could garbage collect (C, F), in the second scenario only B_6 can do so, while B_5 updates its routing table, yielding the correct and desired behavior in both cases.

4.2 Discussion

The example sketched in the previous section should give a feeling of how relocation and adaptation of the delivery paths can be performed in a fully distributed fashion. For lack of space, we have relegated the details of the algorithm formulation into Appendix A. Through the use of administrative control messages, buffering and advertisements, the algoritm makes good use of the already builtin features of Rebeca.

5 Location-Dependent Filters for Logical Mobility

We now describe the algorithmic solution to the scenario where clients are only logically mobile, i.e., they remain attached to a single border broker.

5.1 Main Idea

Consider an arbitrary routing path between a producer (publisher) and a consumer (subscriber). This path consists of a sequence of brokers $B_1, B_2, \ldots, B_{k-1}, B_k$

where B_1 is the local broker of the consumer and B_k is the local broker of the producer (Figure 6 shows the setup for $k = 3$). Assume the consumer has issued a location-dependent subscription F. Using the "usual" content-based routing algorithms, the current value $\tilde{F}$ of F would permeate the network in such a way that the filters along the routing path allow a matching subscription which is published by the producer to reach the consumer. Formally, the filters $F_1, F_2, \ldots, F_k$ along the links between the brokers should maintain a set-inclusion property

$$F_k \supseteq F_{k-1} \supseteq \ldots \supseteq F_2 \supseteq F_1 \supseteq F_0 = \tilde{F}$$

at all times.

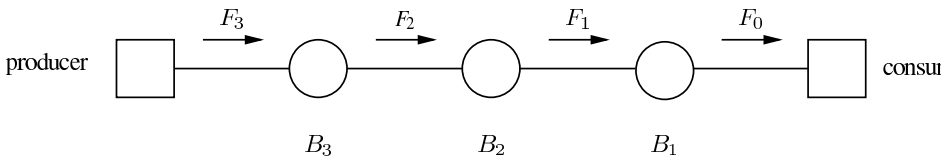

Fig. 6. Network setting for the example.

If F is the only active subscription in the network and if the subscription has permeated the network, the above formula can be simplified to

$$F_k = F_{k-1} = \ldots = F_2 = F_1 = F_0 = \tilde{F}.$$

Obviously, if for any new value $\tilde{F}$ of F a new subscription must flow through the network towards the producers, notifications which are published in the meantime might go unnoticed. The idea of the proposed scheme is to always have the local broker of the consumer do perfect client-side filtering (i.e., set $F_0 = \tilde{F}$), but to let possible future notifications reach brokers which are nearer to the consumer so that their delay to reach the consumer is lower once the consumer switches to a new location.

Let T denote the set of time values, which for simplicity we will assume to be the set of natural numbers $\mathbb{N}$. Let L denote the set of all consumer locations. Then we define a function $loc : T \to L$ which describes the movement of the consumer over time. For example, for a location set $L = \{a, b, c, d\}$ a possible value of loc is $\{(1, a), (2, b), (3, d), \ldots\}$ meaning that at time 1, the consumer's location is a, at time 2 it is b and so on.

We assume that loc is subject to some movement restrictions which in effect define a maximum speed of movement for the consumer. We assume that such a restriction is given by a *movement graph* such as the one depicted in Figure 7. The graph formalizes which locations can be reached from which locations in one movement step of the consumer. One movement step has some application-defined correspondence to one time step.

Given the function loc and a movement graph, it is possible to define a function $ploc : L \times \mathbb{N} \to 2^L$ of possible (future) locations (the notation 2^L

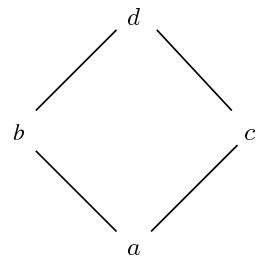

Fig. 7. Movement graph defining movement restrictions of a consumer.

denotes the powerset of L, i.e., the set of all subsets of L). The function takes a current location x and a number of consumer steps $q \geq 0$ and returns the set of possible locations which the consumer could be in starting from x after q steps in the movement graph.

Since a possible move of the consumer always is to remain at the same location, for all locations $x \in L$ and all $q \in \mathbb{N}$ we should require that

$$ploc(x, q) \subseteq ploc(x, q + 1). \tag{1}$$

Taking the example values from above, possible values for *ploc* are as follows:

$$ploc(a, 0) = \{a\} \qquad ploc(a, 1) = \{a, b, c\} \qquad ploc(a, 2) = \{a, b, c, d\}$$

Now, if the consumer is at location a for example, every broker B_i along the path towards a producer should subscribe for $ploc(a, q)$ for some q which is an increasing sequence of natural numbers depending on i and the network characteristics. If the time it takes for a broker to process a new subscription is in the order of the time a client remains at one particular location, then the individual filters F_i along the sample network setting in Figure 6 should be set as $F_i = ploc(a, i)$, e.g., $F_0 = ploc(a, 0) = \{a\}$, $F_1 = ploc(a, 1) = \{a, b, c\}$ and so on. This requirement should be maintained throughout location changes by the consumer. For example, whenever a consumer moves from an old location x to a new location y, the corresponding client node must declare the new location by sending a message to its border broker B_1. This will cause B_1 to change the location-dependent part of filter F_0 for client-side filtering from the old to the new location. Broker B_1 updates its routing table appropriately.

In general, broker B_i sends a message with the new location to B_{i+1} instructing it to change F_i from $ploc(x, i)$ to $ploc(y, i)$ and consequently to update the routing table by removing certain locations and adding new locations. Removing and adding new locations corresponds to unsubscribing and subscribing to the corresponding filters. The normal REBECA administration messages can be used to do this. Note that Equation 1 guarantees the subset relationship which should always hold on every path between producer and consumer.

5.2 Example

As an example, consider the value of *loc* where at time 1 the client is in location a, at time 2 at b and at time 3 at d in the movement graph depicted in Figure 7. Table 1 gives the values of *ploc* for all locations and the first four time instances. For $t = 0$ the value of *ploc* is equal to the current location. For $t = 1$ it returns all locations reachable in one time step in the movement graph, etc.

Table 1. Values of $ploc(x, t)$ for the example setting.

t	$x = a$	$x = b$	$x = c$	$x = d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b.c.d\}$
2	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$
3	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$

Now assume again the setting depicted in Figure 6. The values of Table 1 directly determine the filter settings for $F_0, \ldots, F_3$ as shown in Table 2. At time $t = 1$ the client moves to location b. This means that F_0 changes from $\{a\}$ to $\{b\}$ and that F_1 must unsubscribe to c and subscribe to d, yielding $F_1 = \{a, b, d\}$. At time $t = 2$ the client moves to d, causing F_0 to change to $\{d\}$ and F_1 to unsubscribe to a and subscribe to c. All other filters remain unchanged.

Table 2. Values of filters in example setting.

time t	F_3	F_2	F_1	F_0
0	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c\}$	$\{a\}$
1	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, d\}$	$\{b\}$
2	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{b, c, d\}$	$\{d\}$

The example nicely shows that the method does some sort of "restricted flooding", i.e, all notifications reach broker B_2 but from there the uncertainty is restricted and so is the flow of notifications forwarded by B_2. In fact, the method described above using the *ploc* function can be regarded as an abstraction of both "trivial" implementations discussed in Section 3 (i.e., both implementations are instantiations of our scheme), as we explain in the following section.

5.3 Adaptivity

The example setting above assumes that processing a new subscription by a broker takes about as long as a consumer stays at one particular location. Obviously, it will usually take much less time to process a subscription even if slow or wireless network connections are used (user movement will be in the order

Table 3. Values of $ploc(x,t)$ for trivial *sub*/*unsub* implementation (top) and flooding with client-side filtering (bottom).

$ploc(x,t)$ for global *sub*/*unsub*				
t	$x=a$	$x=b$	$x=c$	$x=d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
2	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
3	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$

$ploc(x,t)$ for flooding				
t	$x=a$	$x=b$	$x=c$	$x=d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$
2	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$
3	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$

of seconds while network delay will be in the order of milliseconds). We now present a scheme which adapts the level of "buffering" in the network to the average movement time of the client. Our algorithm, which for lack of space is detailed in Appendix B.1, satisfies this form of adaptivity.

In the following, we denote the average time a client remains at one location by Δ and the time it takes to process a sufficiently large batch of *sub*/*unsub* messages between brokers B_i and B_{i+1} by δ_i. If the client moves very slowly, meaning that the sum of all δ_i is still less than Δ, we would like the scheme to behave like the trivial *sub*/*unsub* solution. For the example setting from the previous section this would mean that *ploc* has values like in the top part of Table 3. On the other hand, if the client moves very fast and Δ is much smaller than δ_1, the method should revert to flooding (i.e., *ploc* values like in the bottom part of Table 3).

If Δ is neither very large nor very small, what values should *ploc* aquire? The idea is to relate multiples of Δ to the increasing sum of the δ_i as follows: Whenever the sum of δ_i results in a value larger than the next multiple of Δ then the value of *ploc* must "take a step". As an example, assume the following values (all in milliseconds): $\Delta = 100$, $\delta_1 = 120$, $\delta_2 = 50$, $\delta_3 = 50$, $\delta_4 = 20$. Now consider Figure 8 where the sums of these values have been put on a single scale. The *ploc* value for client-side filtering (F_0) is fixed to the current location of the client. Since it takes longer for the brokers B_1 and B_2 to process a location change than the client moves, the system must insert a level of buffering at this point, i.e., *ploc* must cater for one additional step of uncertainty at this stage.

Considering that $\delta_1 + \delta_2 < 2 \cdot \Delta$, a location change can be processed fast enough between B_2 and B_3 so that no additional buffering is necessary at this point. However, the sum $\delta_1+\delta_2+\delta_3 > 2\cdot\Delta$, and so *ploc* must have one additional step between B_3 and B_4. The resulting values in the example setting for *ploc* are shown in Table 4.

Table 4. Values of $ploc(x, t)$ for the example setting with concrete timing values.

t	$x = a$	$x = b$	$x = c$	$x = d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b.c.d\}$
2	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b.c.d\}$
3	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$

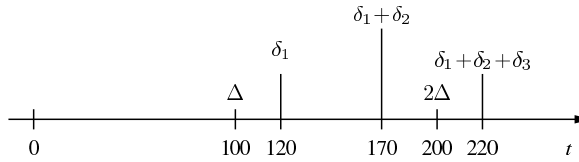

Fig. 8. Estimating *ploc* steps with respect to concrete timing bounds.

5.4 Discussion

The operations "subscribe" and "unsubscribe" in the algorithm refer to operations performed on the original routing table of the corresponding broker. In REBECA, these operations exploit the optimizations of the underlying routing strategy. For example, in covering-based routing subscribing to a filter $\tilde{F}$ may have no effect on the routing table if there already exists a filter F' which covers $\tilde{F}$. The messages about location changes replace the administrative messages which are sent to spread the information about new subscriptions.

From the algorithm it is obvious that the information about a new location-dependent subscription (and about every location change) necessarily permeates the *entire* network of brokers. In the case of a dynamic network environment it is unfortunately not possible to avoid this behavior. To see this, consider a client moving from location x to y and assume that the network behavior is the same in the entire network except that a network link between two very far away brokers B_j and B_i has suddenly become very slow so that it is necessary for B_i to increase the step in the *ploc* function and subscribe to "more" locations than before. But to do this, information about y is needed at B_i.

In networks where all values of δ are fixed and known beforehand, propagation of information about new location dependent subscriptions may cease if there is no change in the set of locations which a broker is subscribed to. In stable networks (where the values of δ do not change so frequently) it may be possible to propagate information and update the "steps" only periodically. In general, it will be advantageous to use *advertisements* which are also provided in REBECA.

We have informally analyzed the total number of messages (notifications and administrative messages) generated by our new algorithm for an arguably realistic network setting, exactly one consumer and two different speeds of consumer movement: fast movement ($\Delta = 1s$) and slow ($\Delta = 10s$). For lack of space, we have relegated a description of the precise system assumptions and details on the derivation of the numbers to Appendix B.2. We compare the results of

these calculations with the total number of messages generated by flooding in Figure 9. It is interesting to see that although our algorithm generates administrative messages on all network links for every location change of the consumer, the fraction of messages saved is still considerable. We also note that many of the assumptions made in calculating these figures have been very conservative. For example, we assume that there is only one consumer in the network and that notifications are generated by the producers according to a uniform distribution over set of locations. Both assumptions prevent the routing strategy optimizations of REBECA to play to their strengths.

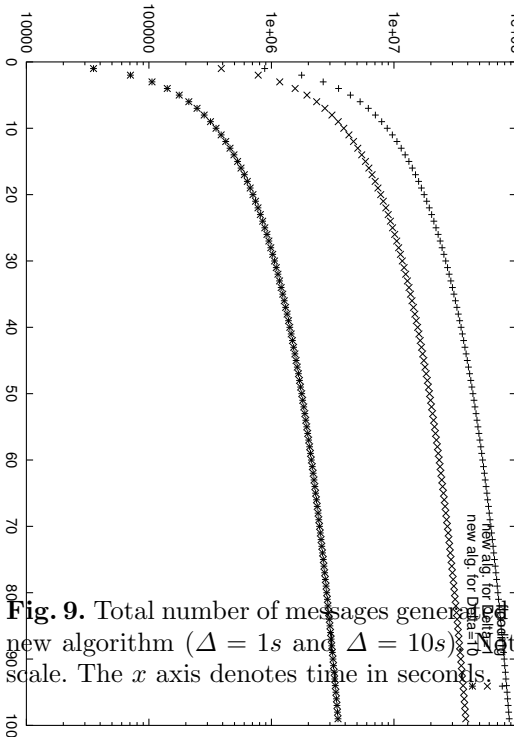

Fig. 9. Total number of messages generated for flooding and two scenarios of the new algorithm ($\Delta = 1s$ and $\Delta = 10s$). Note that the y axis has a logarithmic scale. The x axis denotes time in seconds.

6 Conclusions

This paper has presented an approach to support mobility in existing publish/-subscribe middleware. We have analyzed the problem of mobility from the viewpoint of the event-based paradigm and have identified two separate flavors of mobility. While one flavor is tied to the notion of rebinding a client to different brokers and can be implemented transparently, the other refers to a certain form of location-awareness which offers a client some fine-grained control over notification delivery in the form of location-dependent filters. We have sketched how both notions can be implemented within the existing REBECA event system to exploit its refined routing strategies.

It is quite obvious that even both of our solutions together cannot claim to solve *all* problems related to mobility or together with REBECA constitutes a *complete* mobile computing middleware. In some worst case scenarios both algorithms may lead to undesirable behavior like missing notifications or even starvation of a client, i.e., where a client does not receive notifications due to the latency of the event middleware. For example, this is the case if a client is just too fast for the infrastructure to adapt or if some network links within the broker network are too slow. We have attempted to alleviate these problems by designing adaptive solutions which should work in and can be tuned to most real world scenarios. A detailed analysis of the behavior of the solutions in more extreme and dynamic network settings is a point for future research.

Many other interesting problems concerning the combination of mobility and pub/sub infrastructures remain. For example, designing algorithms and application scenarios for a generalization of location-dependent filters, "dynamic filters", which depend on a function of the local state of the client (not only its current location). As an example, a client might only be interested in receiving notifications for sales which he still can afford.

Acknowledgments

We thank Gero Mühl for his cooperation in the REBECA project and Sidath Bandara Handurukande for helpful comments on an earlier version of this paper.

References

1. J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using events for the scalable federation of heterogeneous components. In P. Guedes and J. Bacon, editors, *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, Sept. 1998.
2. L. Capra, W. Emmerich, and C. Mascolo. Middleware for mobile computing (a survey). Research Note RN/30/01, University College London, July 2001.
3. A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in supporting event-based architectural styles. In *Proceedings of the Third International Workshop on Software Architecture (ISAW '98)*, pages 17–20, Orlando, FL, USA, 1998.
4. 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, 2001.
5. G. Cugola and E. Di Nitto. Using a publish/subscribe middleware to support mobile computing. In *Proceedings of the Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, Nov. 2001.
6. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *Proceedings of of the 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, FL, USA, Nov. 1998. ACM Press.
7. P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. In L. Northrop and J. Vlissides, editors, *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*, pages 254–269, Tampa Bay, FL, USA, 2001. ACM Press.

8. L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.
9. L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 385–392, Madrid, Spain, 2002. ACM Press.
10. L. Fiege, G. Mühl, and F. C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 2003. to appear.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.
12. 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 (MobiDE01)*, Santa Barbara, CA, May 2001.
13. Y. Huang and H. Garcia-Molina. Publish/subscribe tree construction in wireless ad-hoc networks. In M.-S. Chen, P. Chrysanthis, M. Sloman, and A. Zaslavsky, editors, *4th International Conference on Mobile Data Management (MDM 2003)*, volume 2574 of *LNCS*, pages 122–140, Melbourne, Australia, 2003. Springer-Verlag.
14. IBM. Gryphon: Publish/subscribe over public networks. Technical report, IBM T. J. Watson Research Center, 2001.
15. H.-A. Jacobsen. Middleware services for selective and location-based information dissemination in mobile wireless networks. In *Proceedings of the Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, Nov. 2001.
16. D. Johnson. Scalable support for transparent mobile host internetworking. *Wireless Networks*, 1:311–321, Oct. 1995.
17. N. Lynch. Supporting disconnected operation in mobile CORBA. MSc thesis, University of Dublin, Trinity College, Sept. 1999.
18. G. Mühl. Generic constraints for content-based publish/subscribe systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *LNCS*, pages 211–225, Trento, Italy, 2001. Springer-Verlag.
19. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.
20. G. Mühl and L. Fiege. Supporting covering and merging in content-based publish/-subscribe systems: Beyond name/value pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.
21. G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In A. Boukerche and S. Majumdar, editors, *The Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, TX, USA, October 2002. IEEE Press.
22. 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, Dec. 1993. ACM Press.
23. W. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia, September 1997.*, 1997. http://elvin.dstc.edu.au/doc/papers/auug97/AUUG97.html.
24. P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness – transparent information delivery for mobile and invisible computing. In *First International Symposium on Cluster Computing and the Grid*, pages 277–287, Brisbane, Australia, May 2001. IEEE/ACM.

A Algorithm for Roaming Clients

We now give the complete details of the algorithm for roaming clients which was presented by example in Section 4. The algorithm itself is given in two figures:

- Fig. 10 gives the algorithm for a border broker directly connected to a moving client
- Fig. 11 gives the algorithm for an inner broker made aware of moving clients and receciving messages from neighboring brokers.

A.1 Basic Case Analysis

By design, the mechanism we use introduces a natural way of distributed caching, which seems in general preferable to a potentially problematic central caching proxy. Hence, for every border broker connected to a client a caching data structure must be added.

We now analyse the different cases which the algorithm caters for and relate them to the standard behavior of the underlying REBECA system. The algorithm has to distinguish between four important situations:

1. A completely new client "wakes up" for the first time and submits some subscription to a broker B for the first time. This is slightly different from a new subscription issued after B has already received some other subscription from this client (i.e., knows already that the client is present).
2. The client was suspended (e.g., for saving energy) and has now resumed operation and wants to be set up properly by receiving events it has missed in the meantime but no location change occurred.
3. A client X connecting to a border broker B_{new} is a roaming client moving from one location to another (i.e., sending a subscription (X, F) with a sequence number of the last notification received for this subscription to broker B_{new}). Obviously, this has to be done whenever the client detects the change of context[2].
4. A client is suspended at one location and then moved to another location where it resumes operation. This situation is analog to a combination of the situations above and has not to be handled separately.

A.2 Algorithm Behavior

Given the four situations above, we now explain how the algorithm reacts.

[2] Please note that this is a rather complicated situation for an event middleware: as a mobile client usually cannot predict a change of broker before leaving its range (e.g., because it is just leaving a wireless network cell) it can only react to the new situation. For a "roaming-aware" client this means that it cannot unsubscribe to a producer at the old border broker before connecting to a new one. Hence, mobility in a pub/sub system is more likely a sequence of subscribe operations than a sequence of subscribe-unsubscribe-subscribe operations.

Wakeup of a new client. This case is already covered by the standard REBECA middleware: The subscription is added to the local routing table and the subscription (X, F) is forwarded in the direction of all known advertisements matching filter F. As a completely new subscription starts with the sequence number $num = 0$, the broker can separate this situation from situation 3. For details refer to the first block of statements of Fig. 10.

Resuming of a client at same border broker. This case is also covered by the standard event mechanisms provided by REBECA, as the broker knows the client and its subscriptions (the broker has some routing table entries for the client) and therefore event delivery simply could be resumed. Nevertheless, to adhere to the requirement of completeness, the broker has to instantiate a *cache* for each client and subscription (we use a ring buffer data structure for enforcing the maximum number of cached notifications together with a Time-to-Live (TTL) mechanism for optimizing the utilization of buffer space). Whenever the client gets online again it is oblieged to re-issue its subscription together with the last sequence number it has received from a broker. The broker simply has to send all cached notifications to the client beginning with the next notification in the sequence (see first block of statements in Fig. 10).

Direct relocation. The main issues of the relocation process are: (i) tracking down the old location without explicit knowledge, as neither client nor new broker know the old broker's identifier, (ii) fetching any cached notification for the client from there, and (iii) the old broker has to receive an explicit sign-off for garbage collection. We have to distinguish between border and inner brokers for this case:

1. Border broker at new location:
   This is comparable to situation 2 above together with a new flag on a routing table entry (*inactive*) initiating buffering until further notice (see second and third block of statements in Fig. 10). Upon reception of a *replay* message, the cached notifications will be appended to the replayed ones and then delivered.
2. Any inner broker:
   For any given relocation process an inner broker can play one of three possible roles, either it is an ordinary broker on the new path from the producer to the consumer and has not encountered this particular subscription before, or it is the one broker "sitting" on the junction between the old and the new path, or one of the brokers on the path to the old location, somewhere between the broker at the junction and the border broker at the old location.
   - Ordinary broker:
     The same as every inner broker for situation 1 (see also first block of statements in Fig. 11).
   - Broker at a junction:
     This is a broker sitting on the junction between old and new path on

the itinerary to a client[3]. The broker can determine this state for any incoming subscription by inspecting the routing table. Whenever a given subscription is already present, the broker is the first one on the path to the old location (with respect to the propagated subscription). We introduced a new inter-broker message type $fetch(X, F, num, B_i)$ (piggybacked to a normal subscription) to handle this situation. The message is sent along the old path for two reasons: (i) to initiate sending of cached notifications at the old location, and (ii) to determine the number of obsulete hops on the path which can be discarded after the client has left. Finally, the broker updates its routing table to point to the new location and resumes normal operation.

- Broker on the old path receiving a *fetch*:
  Whenever an inner broker receives a fetch request (see second block of statements in Fig. 11) obviously the broker is a hop on the path between the first junction and the old location of client X. It has to send the message further along the path after checking whether or not it has a path to another producer the client was subscribed to (multiple producer) and (i) if not, to simply update its routing table to point to the broker the message was received from or (ii) if so, to replace the broker identifier in the *fetch* request with its own, to indicate that the path to client X is not obsolete from this broker on and then to update the routing table like in (i).
- Broker on the old path receiving a *replay*:
  Whenever an inner broker receives a *replay* message[4] it simply routes the message towards the new client location after checking whether or not it can discard the routing table entry for client X. To do so it checks the broker identifier B_{prod} enclosed in the message: (i) the identifier is not the same as the one of the current broker, i.e., the current broker is located some hops after the last junction, or (ii) the identifier is identical, i.e., this broker is located at the last junction. In case of (i) the routing entry can be discarded and in case of (ii) the broker identifier in the replay message is set to some well known value *nil* indicating that the process of freeing resources has terminated.

3. Border broker at old location receiving a *fetch* message:
   Whenever a border broker receives a $fetch$ message for some (X, F, num) from broker B_j it has to create a new replay message with all buffered notifications for (X, F) starting from $num + 1$ and send it in the direction of B_j. After this it can clean up und de-allocate all resources allocated to client X, i.e., has received an equivalent to an explicit "sign-off".

[3] Note that such a broker always exists on the path to a particular producer due to the structure of the broker network as defined in section 2.2.

[4] This message type is part of the standard REBECAsystem and in this context indicates that the relocation process is finishing and on the "way back" to the new location of client X.

Once the complete relocation process has terminated the algorithm asserts the requirements stated above, namely that (i) all notifications send to the old location (modulo messages discarded due to limited caches or expiration periods) will eventually reach broker B_{new}, (ii) broker B_{new} delivers all notifications to client X in correct order, i.e., respecting sender FIFO by sending redirected notifications first and then newly arrived ones, (iii) broker B_{old} at the old location will eventually detect the relocation of client X and can discard all subscriptions and caches for this client, i.e., has a simple and elegant criteria for garbage collection.

Client move after suspension. As the new broker cannot distinguish between a client newly powered on or one entering its range, this situation is already covered by situation 1, 2, and 3.

```
{ upon receiving of subscription (X,F,num) from X do }
  if ((X,F) not in ldfTable) then {
    allocate new entry in ldfTable
    initialize entry with (F,X,B_j)
    initialize cache(X,F)
    if (num > 0) then {
       set blocking flag on entry (F,X,B_j) in ldfTable
    }
    send (X,F,num) to all neighboring brokers we
         have received a matching advertisement from
  } else {
    send cache(X,F) to X
  }

{ upon receiving of fetch(X,F,num,B_prod) from B_j do }
  if (#Advertisements matching Filter F > 1) then {
     update entry in ldfTable to (F,X,B_j)
  } else {
     delete entry in ldfTable with (X,F,*)
  }
  send replay(fetch(X,F,num,B_prod),[e_1,...,e_n]) to B_j

{ upon receiving of replay(fetch(X,F,num,nil),[e_1,...,e_n])
  from B_j do }
  prepend notifications [e_1,...,e_n] to cache(X,F)
  unset blocking flag on entry (F,X,B_j) in ldfTable
  send all notifications to client X
```

Fig. 10. Actions of a border broker receiving a message from a client X

```
{ upon receiving of subscription (X, F, num) from B_j do }
if ((X, F) not in ldfTable) then {
    allocate new entry in ldfTable
    initialize entry with (F, X, B_j)
    send (X, F, num) to all neighboring brokers we
         have received matching subscriptions from
} else {
    broker_oldnext = get(ldfTable, X, F)
    update entry in ldfTable to (F, X, B_j)
    send fetch(X, F, num, B_i) to broker_oldnext
}

{ upon receiving of fetch(X, F, num, B_prod) from B_j do }
broker_oldnext = get(ldfTable, X, F)
update entry in ldfTable to (F, X, B_j)
if (#Advertisements matching Filter F > 1) then {
    send fetch(X, F, num, B_i) to broker_oldnext
} else {
    send fetch(X, F, num, B_prod) to broker_oldnext
}

{ upon receiving of replay(fetch(X, F, num, B_prod), [e_1, ..., e_n])
from B_j do }
broker_next = get(ldfTable, X, F)
if (B_prod != nil) then {
    if (B_prod == B_i) then {
        send replay(fetch(X, F, num, nil), [e_1, ..., e_n]) to broker_next
    } else {
        delete entry in ldfTable with (X, F, *)
        send replay(fetch(X, F, num, B_prod), [e_1, ..., e_n]) to broker_next
    }
}
```

Fig. 11. The algorithm for an inner-network broker receiving a message from broker B_j.

B Logical Mobility: Algorithm and Analysis Details

B.1 Algorithm

For some client X we are given a set of locations L_X and a movement graph $M_X(L_X, E_X)$. From M_X calculate the function $ploc_X : L_X \times \mathbb{N} \rightarrow 2^{L_X}$. This can be done at setup time of the pub/sub system. We assume that all brokers know $ploc_X$.

Data structures. Every broker has an additional routing table data structure *ldfTable* where location dependent filters are stored in their original form, i.e., with the marker *myloc* uninterpreted. In general, we assume that $F_X \equiv V, (location \in \lambda)$ where V is an arbitrary sequence of name/value pairs without references to *myloc*. We instantiate F into $\bar{F}$ by simply replacing λ with some set of locations λ'. We denote this instance as $F[\lambda/\lambda']$.

For every entry in *ldfTable* the originating client is stored (i.e., X) and a natural number *oldstep*.

Every broker B_i maintains a network statistic about the average network delay $\delta_{B_i \rightarrow B_j}$ for sending messages from B_i to B_j. Every client X maintains a statistic of the average time Δ he remains at a specific location.

```
{ upon receiving new location-dependent subscription F
    from X with current location x and given Δ do }
  allocate new entry e in ldfTable
  initialize e with F, X, and 0
  subscribe to F[λ/{x}]
  send (X, x, Δ, 0) to all neighboring brokers

{ upon notification that client moves from location x to y
    given Δ do}
  for all filters F in ldfTable referring to X do
      unsubscribe to F[λ/ploc_X(x, 0)]
      subscribe to F[λ/ploc_X(y, 0)]
      send (X, x, y, Δ, 0) to all neighboring brokers.

{ upon receiving an unsubscription for location-dependent filter
    from X at location x do }
  unsubscribe to F[λ/{x}]
  de-allocate entry containing F in ldfTable
  send (X, x, F) to all neighboring brokers
```

Fig. 12. Algorithm for the local broker B_X of client X.

```
{ upon receipt of (X, x, Δ, dist) from B_j do }
  dist := dist + δ_{B_i→B_j}
  step := ⌈dist/Δ⌉
  allocate new entry in ldfTable
  initialize entry with F, X, and step
  subscribe to F[λ/ploc_X(x, step)]
  send (X, x, Δ, dist) to all neighboring brokers except B_j

{ upon receipt of (X, x, y, Δ, dist) from B_j do}
  dist := dist + δ_{B_i→B_j}
  for all filters F in ldfTable referring to X do
      newstep := ⌈dist/Δ⌉
      unsubscribe to F[λ/ploc_X(x, oldstep)]
      subscribe to F[λ/ploc_X(y, newstep)]
      oldstep := newstep
      store oldstep in ldfTable
      send (X, x, y, Δ, dist) to all neighboring brokers except B_j

{ upon receiving (X, x, F) from B_j do }
  unsubscribe to F[λ/ploc(x, oldstep]
  de−allocate entry containing F in ldfTable
  send (X, x, F) to all neighboring brokers except B_j
```

Fig. 13. Algorithm for the broker B_i receiving a message from broker B_j.

Algorithm. The algorithm for a local broker B_X of some client X is depicted in Figure 12. If a client issues a new location-dependent subscription F it is entered into the table *ldfTable* and the local routing table is updated by subscribing to the proper instance $\tilde{F}$. Information about this new filter is forwarded through the network within the algorithm for the other brokers (see Figure 13). During this process, all other brokers allocate an appropriate entry in their *ldfTable* and subscribe to the "right" instance of F given the current distance from X. Note that during the propagation of the new filter through the network the value of *dist* continuously sums up the network delay along the path. This value determines the "step" of the *ploc* function which is used to instantiate F correctly.

When a client changes location from x to y, all the brokers similarly update their routing tables by taking the information about the changed location, unsubscribing to the old filter and subscribing to the new correct instance of F. In doing so, the distance *dist* is recalculated and may also lead to changes in how F is instantiated.

B.2 Informal Analysis

We now give the details of our the quantitative analysis summarized in Section 5. The main question we posed was how much network traffic our algorithm can save compared to flooding. The answer to this question depends on many different parameters and a complete analysis of all parameters is still out of the scope of this paper. Instead, we calculate the total number of messages processed for a set of common network scenarios and derive some conclusions from these numbers.

The base scenario we consider consists of a pub/sub system which is built around a backbone of event brokers. The brokers within the backbone are connected with high speed communication links on which the network delay δ_f is very low. We assume that clients and their local brokers reside on the same physical process so that communication between them is instantaneous. However, local brokers are attached to the backbone by very slow communication links that have a high network delay δ_s. In our case we chose $\delta_f = 10ms$ and $\delta_s = 600ms$ and assume that they do not change.

To ease calculations, the broker backbone is assumed to have the structure of a tree with degree b and h levels (see Figure 14). Each local broker serves p clients which will play the role of notification producers in this scenario. In our case we set $b = 3$, $h = 4$ and $p = 10$. Because of the tree structure, we can calculate the total number of brokers n_b as $1 + \sum_{i=0}^{h} b^i$.

There is exactly one consumer in the system which is attached to a local broker at the root of the broker tree. This consumer issued a location-dependent subscription. We assume that the size of the set L of possible locations is 100 and that the movement graph of the consumer is "geographic", i.e., with every possible step of the client the number of possible locations is multiplied by some factor s. In our case, we assume $s = 4$. We assume that the user changes location every Δ seconds.

Producers generate r notifications per second which are uniformly distributed over the set L of possible locations.

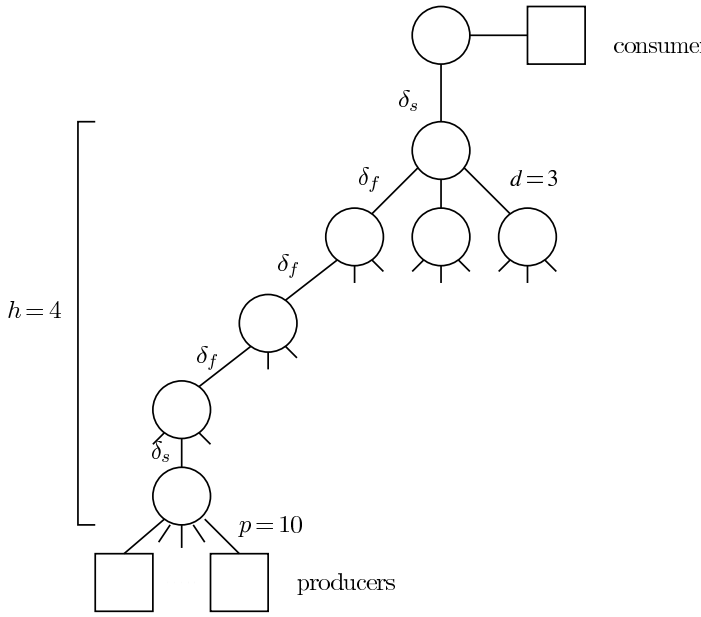

Fig. 14. Sample scenario analyzed.

For every particular choice of values for these parameters, we calculate two values:

- $N_f(t)$: the total number of messages generated in the network if flooding is used as basic routing technique.
- $N_n(t)$: the total number of messages (notifications and control messages) generated in the network if our new algorithm for location-dependent filters is used.

We calculate these values for two different types of user movement: slow movement ($\Delta = 10s$) and fast movement ($Delta = 1s$).

For flooding, the user movement is unimportant in the calculation of the total number of messages processed over time. Let n_{lb} denote the number of local brokers in the system belonging to producers. Due to the tree structure of the system, $n_{lb} = b^h$. Since every local broker has p clients (producers) attached to it, the total number of producers n_c in the system calculates to $n_c = n_{lb} \cdot p$. Every producer emits r notifications per second. Hence, the number of notifications produced per second $n_{ps} = n_c \cdot r$. Since every produced message has to cross every link, we can calculate

$$N_f(t) = n_{ps} \cdot n_{links} \cdot t$$

where, due to the tree structure, the number of links between brokers is $n_{links} = n_b - 1$.

The calculation of a similar formula for our algorithm depends on Δ. With $\Delta = 1s$ and due to the values for δ_s and δ_f, our algorithm must "buffer" one step of the movement graph per slow link, i.e., the local broker of the consumer receives all notifications which are one step away from the current locations (i.e., a fraction of $s/|L|$ of all produced notifications). Also, the broker following the local broker of the producer must receive all notifications two steps away of the current consumer location (i.e., a fraction of $s^2/|L|$ of all produced notifications). In the case $\Delta = 10s$ the latter buffering is not necessary.

For $\Delta = 10s$ we end up with

$$N_n(t) = n_{ps} \cdot n_{links} \cdot \frac{s}{|L|} \cdot t + n_{links} \cdot \lfloor \frac{t}{10} \rfloor$$

where the first part of the sum corresponds to the restricted flooding (the fraction of $s/|L|$ messages crosses every link) and the second part corresponds to the control messages introduced by our algorithm. Since control messages about location changes flood the network, their number must be treated like in flooding. However, a new control message is generated only once in $\Delta = 10s$.

For $\Delta = 1s$, we must take into account that a fraction of $s^2/|L|$ notifications crosses the first link from the local broker to the next broker in the network. Hence, we multiply every produced message with this fraction together the number of links starting from local brokers (n_{lb}). To this we add the number of remaining links $n_{rest} = n_{links} - n_{lb}$ times the original fraction of $s/|L|$ of produced messages. The second part of the sum is again to account for the control messages, which flood the network once per second.

$$N_n(t) = \left[n_{ps} \cdot n_{lb} \cdot \frac{s^2}{|L|} + n_{rest} \cdot n_{ps} \cdot \frac{s}{|L|}\right] \cdot t + t \cdot n_b$$

The results of our calculations are depicted in Figure 9 (note that the y axis has a logarithmic scale). It shows that for the given settings and the two different scenarios the new algorithm imposes a smaller number of total messages than flooding. The graph confirms that a slower user movement can save more messages. In Figure 9, the values underlying the computation are $\delta_s = 600ms$, $\delta_f = 10ms$, $b = 3$, $h = 4$, $p = 10$, $|L| = 100$, $r = 1$, $s = 4$, $n_b = 364$, $n_{lb} = 243$, $n_c = 2430$, $n_{ps} = 2430$, $n_{links} = 363$, $n_{rest} = 120$.