Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications

Mauro Caporuscio,[†] Antonio Carzaniga,[‡] and Alexander L. Wolf [‡]

[†]Dipartimento di Informatica
Universita' dell'Aquila
via Vetoio, 1 67100 L'Aquila, Italy
caporuscio@di.univaq.it

[‡]Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado, 80309-0430 USA
{carzanig,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-944-03 January 2003

Abstract

This paper presents the design and evaluation of a support service for mobile, wireless clients of a distributed publish/subscribe system. A distributed publish/subscribe system is a networked communication infrastructure where messages are published by senders and then delivered to the receivers whose subscriptions match the messages. Communication therefore does not involve the use of explicit addresses, but rather emerges from the dynamic arrangement of publishers and subscribers. Such a communication mechanism is an ideal platform for a variety of Internet applications, including multi-party messaging, personal information management, information sharing, on-line news distribution, service discovery, and electronic auctions. Our goal is to support such applications on mobile, wireless host devices in such a way that the applications can, if they chose, be oblivious to the mobility and intermittent connectivity of their hosts as they move from one publish/subscribe access point to another. In this paper we describe a generic, value-added service that can be used in conjunction with publish/subscribe systems to achieve these goals. We detail the implementation of the service and present the results of our evaluation of the service in both wireline and emulated wireless environments.

1 Introduction

A publish/subscribe system is a middleware communication service that delivers messages from a sender to one or more receivers using the preferences expressed by those receivers, rather than relying on an explicit destination address set by the sender. Specifically, a sender *publishes* messages, while receivers *subscribe* for messages that are of interest to them; the system is responsible for delivering published messages to matching subscribers. Examples of Internet applications that can use a publish/subscribe system are multi-party messaging, personal information management, information sharing, on-line news distribution, service discovery, and electronic auctions.

A publish/subscribe system can be implemented by a centralized server or by a network of message routers. In the first case, every client application—acting as a publisher, a subscriber, or both—uses the system by connecting to a single server, which then acts simply as a switch. In the second case, illustrated in

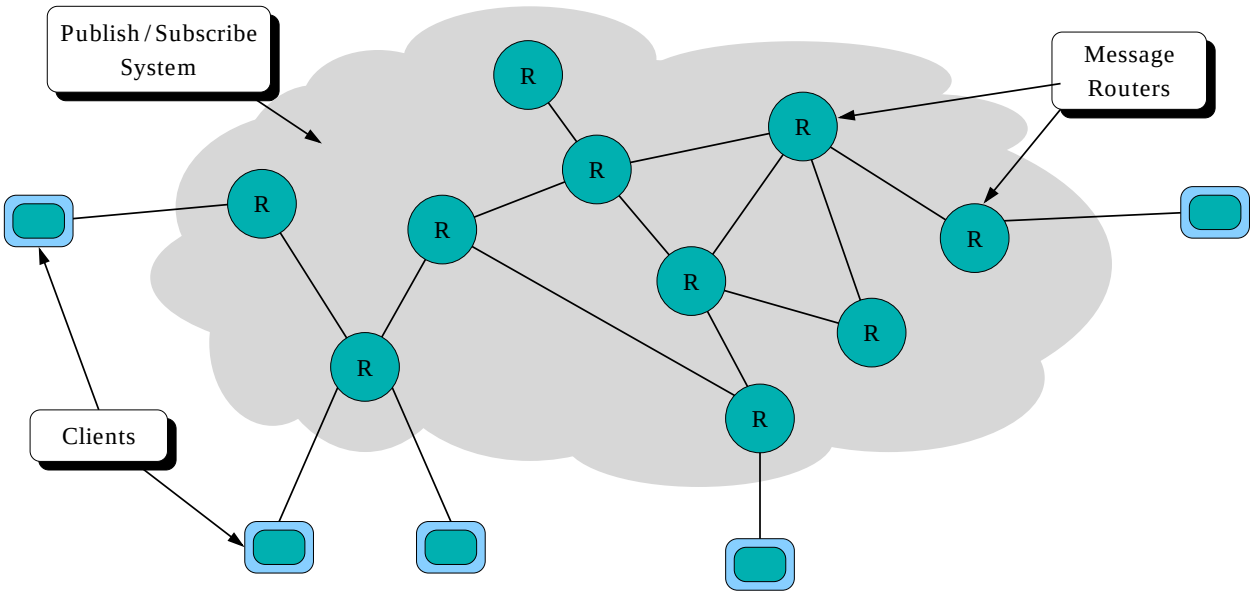

Figure 1: Distributed Publish/Subscribe System

Figure 1, clients connect to one of several distributed access points. The access points are themselves interconnected through message routers that cooperate to form a distributed, coherent communication service.

In this paper, we focus on this latter type of publish/subscribe implementation. In particular, our goal is to support mobile client applications, that is, applications that move from one access point to another during their execution. We do this by implementing a *mobility support service* that transparently manages active subscriptions and incoming messages when an application detaches from one access point until it reattaches at another.

In order to better scope our work, it is useful to distinguish two classes of mobile publish/subscribe applications. The first class consists of applications running on a mobile host such as a notebook computer, a PDA, or a cellular phone. As the host moves (along with the person using the host), the applications may access the network from various locations through the publish/subscribe system. The host may also be disconnected for some significant amount of time while moving from one access point to the other. The duration of these black-out periods, and the physical and topological distance between access points, may vary from local-area networks to wide-area networks, depending on the specific network technology in use, and obviously depending on the specific movements of the user. The second class of mobile applications consists of those that, using some form of mobile-code technology [12, 24], migrate autonomously from one host to another. During their life time, these applications (often referred to as "mobile agents") may execute on several hosts, and on any given host they may bind to and access the publish/subscribe system.

Although both classes of applications may benefit from using our mobility support service, the work

described in this paper focuses primarily on the first class of applications, that is, on applications resident on the device carried by a mobile user. Our choice of scope is based on two practical observations: first and foremost, these applications are by far the most prevalent and, second, their mobility needs are naturally limited to the mobility of the person using them, and therefore occur at a manageable time scale.

The contributions of this paper are a detailed description of a portable, adaptive mobility support service for distributed publish/subscribe systems and an extensive evaluation of that service. The design is based on a client proxy that acts as an interface to the publish/subscribe system while the client is disconnected, and that switches subscriptions and messages from one access point to the other when the client reconnects to the network. In addition to a basic buffering function, the proxy implements synchronization mechanisms designed to reduce the loss or duplication of information during the switch-over operation. These mechanisms are portable in the sense that they do not rely on internal operations of any specific publish/subscribe system. Nevertheless, they are also adaptive in the sense that they implicitly exploit certain features of the underlying system opportunistically.

At a high level, our evaluation has three components. In the first part, we implemented the mobility support service on top of three different distributed publish/subscribe systems. The goal of this part of the evaluation was a feasibility demonstration of the portability of the service architecture. (A secondary outcome is a revealing survey, summarized in Table 1 of Section 2, of the architecture and availability of several publish/subscribe systems.) In the second and third parts of the evaluation, we analyzed the performance of the three implementations in a variety of scenarios. Specifically, the goal of the second part was to compare the behavior of the implementations on a wireline network and on a GPRS [1] wireless network. This evaluation is particularly relevant to the environment targeted by our work. The goal of the third part was to assess the effectiveness of the synchronization mechanisms across the various implementations and under different workloads of subscriptions and notifications.

Overall, our results indicate that the mobility service architecture we propose is portable, and that its optional synchronization mechanisms adapt nicely to a significant sample of implementation features of the target publish/subscribe system.

The paper is organized as follows. Section 2 discusses related work, including the results of our survey of publish/subscribe systems. Section 3 details the architecture and implementation of the mobility support service. Section 4 reports the results of the performance evaluation. Finally, Section 5 summarizes our experience and discusses future work.

2 Related Work

The work presented in this paper is clearly rooted in our previous and ongoing study of distributed publish/subscribe systems [6]. The idea of a publish/subscribe system is quite mature, and a fair number and variety of publish/subscribe systems have been proposed and described, including research prototypes [2, 10, 18, 19], commercial products [23], and attempts at standardization [8, 9, 21, 22]. In recent years, researchers have focused on techniques to implement such systems with scalable, distributed architectures. However, despite this interest in distributed architectures, the issue of mobility or relocation of clients has not been explored in great detail.

The general problem of dealing with mobile clients is not specific to publish/subscribe systems. In fact, the designer of almost any distributed system must deal with duplicate or lost information when allowing clients to move or simply to switch their point of contact. A common case is that of a person changing addresses (e-mail or even postal address.) In this situation, a common approach would be to announce the change of address to every known sender, and to maintain both addresses for some period of time to catch messages inadvertently or unknowingly sent to the old address, possibly redirecting their senders to the new address. A more formalized, but conceptually identical, mechanism is used to implement network-level mobility support in mobile IP [4]. Our approach differs from this class of solutions because we do not maintain a fixed proxy, but rather combine a place-holder proxy with a reconfiguration of the publish/subscribe system to adapt to the new location.

To our knowledge, the first publish/subscribe system to propose explicit support for mobile clients was JEDI [10]. JEDI defines two functions—*move-out* and *move-in*—that a client can use to explicitly detach from the publish/subscribe network, and to reconnect to it, possibly at a different location. In our mobility

support system we have adopted the same move-out/move-in interface. As for the implementation of the move-out and move-in functions, the authors of JEDI briefly discuss how a client can move by detaching, serializing its state, and reconnecting. However, they do not detail the effect of a movement on the publish/subscribe system. Also, they recognize the synchronization problems involved in the movement, but explicitly avoid exploring those issues.

Other researchers have studied the problem of mobility in the context of publish/subscribe systems. Among these studies, some are concerned with the general requirements posed by collaborative applications and mobile clients over the publish/subscribe system. The scope of this type of work is therefore broader than the one we describe in this paper, and deals with other publish/subscribe interface design issues, such as the data definition for publications and the corresponding query language used to define subscriptions. One such study is that of Fenner et al. [11].

In other cases, researchers have focused on the specific problem of enhancing connectivity with mobile clients, which is also the focus of this paper, and have proposed techniques to solve this problem. The work of Huang and Garcia-Molina [13] belongs to this category. They propose a general framework, summarizing previous research of their own and of others, to adapt centralized and distributed publish/subscribe systems to a mobile environment. Although the proposed adaptation strategies are, at a high level, similar to the ones we present in this paper, our work differs in two ways from that of Huang and Garcia-Molina. First, their general approach is to modify the publish/subscribe system to adapt its functionality to the mobile environment, while our goal is to implement a mobility support service that is independent from the underlying publish/subscribe system. Second, they do not present an implementation of their techniques nor any empirical analysis of their performance, while a large part of our work focuses on experimentation with existing publish/subscribe systems.

	Architecture	**Software Availability**
Elvin	Distributed	Available
Herald	Distributed	Not Available
Hermes	Distributed	Not Available
JEDI	Distributed	Not Available
JMS FioranoMQ	Distributed	Available
JMS JBossMQ	Not Specified	Not Available
JMS Joram	Distributed	Available
JMS OpenJMS	Centralized Distributed	Available Not Available
CORBA OmniNotify	Centralized	Available
CORBA OpenORB	Centralized	Available
Siena	Distributed	Available
STEAM	Distributed	Not Available

Table 1: Survey of Publish/Subscribe Systems.

Another large body of related work consists of the general literature on publish/subscribe systems. Although a comprehensive review of the field is outside the scope of this paper, below we present a brief survey covering a number of representative publish/subscribe systems. The goal of the survey is to motivate our choice of target platforms in our implementation and experimentation activities. Therefore, the survey is not based on general technical evaluation criteria, such as interface capabilities, performance, scalability, etc., but instead it is limited to two basic questions: First, does the system provide a truly distributed implementation consisting of multiple, interconnected elements or just a single, centralized server? Second, is the implementation available for download and use? Table 1 summarizes our findings.

2.1 Elvin

Elvin is an event notification system developed by the Distributed Systems Technology Center (DSTC) at the University of Queensland [19]. Its main features are quenching, server discovery, server clustering, and

federation. Quenching allows producers to receive information about consumer requests so that they can limit their output to only those events that are of interest to at least one client.

Elvin clients connect to one Elvin server. A client can dynamically discover the servers available on a local-area network. Elvin allows two levels of distribution. Servers collocated on a local-area network can be grouped into a *cluster*. Clusters appear as a single access point to clients, and use multiple servers to perform some load balancing. Clusters and servers can also connect to each other across a wide-area network to form a *federation*. The interconnection of individual servers and clusters in a federation must be acyclic. Routing within a federation is statically configured, meaning that every component of a federation must be statically configured for sharing specific notification messages and subscriptions.

The Elvin client API and server implementation are available for download and use.[1]

2.2 Herald

Herald [3] is a project of Microsoft Research. It is described as a publish/subscribe event notification service, deployed as a self-configuring federation of peers. Herald defines three main actors: *publishers*, *subscribers*, and *rendezvous points*. Rendezvous points act as service access points for clients, meaning that clients can communicate with other clients connected to the same access point. It is not clear, however, whether or how multiple rendezvous points can be interconnected to form a single, distributed service implementation. Herald offers no filtering capability, meaning that it "delegates" the selection of events of interested to clients.

We were unable to find an implementation of Herald.

2.3 Hermes

Hermes [18] is a publish/subscribe system developed at the Computer Laboratory at the University of Cambridge. It consists of two main components: *event clients*, which can be both publishers and subscribers, and *event brokers*. Event brokers represent the access points and provide a distributed implementation of the service required by the clients. Brokers, which may be interconnected with each other in an arbitrary topology, route messages through the broker network depending on their content and the set of current subscriptions.

Hermes promises to implement a truly dynamic routing strategy for subscriptions and publications, and therefore seems like a good candidate target for our mobility service, however we were unable to find an implementation.

2.4 JEDI

JEDI [10] is an event-based infrastructure designed at CEFRIEL and Politecnico di Milano. It is based on the notion of *active objects*—publishers and subscribers—exchanging messages through an *event dispatcher*. The event dispatcher is a logically centralized component that can be implemented by a distributed set of *dispatching servers* connected in an acyclic topology. JEDI offers native support for client mobility through its *moveIn* and *moveOut* operations. A client may disconnect from its server by calling moveOut. When a client is disconnected, the event dispatcher stores all the events in which the client is interested. The client can then reconnect to the same dispatcher or to another dispatcher by calling moveIn. The effect of the moveIn operation is to retrieve all events buffered by the server.

We were unable to find a detailed specification of the moveIn and moveOut operations implemented in JEDI. In any case, JEDI is not available for download.

2.5 Java Message Service

The Java Message Service (JMS) [22] is an API specification defined by Sun that allows applications to create, send, receive, and read messages. JMS supports both a point-to-point and a publish/subscribe approach to messaging, and defines a separate domain for each one. In JMS, communication can be loosely coupled, asynchronous, and reliable. In order to provide reliability, JMS provides a particular kind of subscriber,

[1] http://elvin.dstc.edu.au/

called a *durable subscriber*. Durable subscriptions allow an application to receive messages published while it is inactive. Messages published while a durable subscriber is disconnected are stored by JMS and delivered when the subscriber becomes active.

We looked at a few implementations of JMS, including JBossMQ,[2] OpenJMS,[3] Joram,[4] and FioranoMQ.[5] All are available, but we chose to focus on FioranoMQ, since it seems to be the most complete and functional.

2.5.1 FioranoMQ

FioranoMQ is an implementation of JMS. It implements the entire of JMS API, and provides proprietary features for management, data storage, and load balancing. FioranoMQ is designed to have a centralized architecture as well as a distributed architecture made up of a federation of servers. A federation is built by using an additional entity called a *repeater*. The FioranoMQ repeater enables the communication between servers by using the publish/subscribe model. This means that a repeater is a dedicated JMS client that acts as a gateway between servers. As a JMS client, the repeater can be configured as a normal subscriber as well as a durable subscriber (see above). This ensures server-to-server communication in case of network failure.

FioranoMQ is available for download and use for an evaluation period.

2.6 CORBA Notification Service

The CORBA Notification Service (NS) [9] is the Object Management Group's specification of a publish/subscribe messaging system. Conceptually similar to JMS, NS extends the previous Event Service (ES) [8]. ES specifies a simplistic channel-based publish/subscribe service. NS adds some advanced capabilities, including a type structure for notifications and additional content-based filtering performed in addition to the underlying channel-based delivery mechanism. Logically, NS is made up of three components: an *event consumer*, an *event supplier*, and *event channels*, which broker events.

Just like JMS, NS is only an API specification. Among the CORBA implementations that implement NS, we reviewed OpenORB[6] and OmniNotify.[7] Neither of these implementations provides a distributed architecture.

2.7 Siena

Siena [6] is a distributed publish/subscribe system developed at the University of Colorado. Siena's goal is to maximize both the expressiveness of the subscription filtering language, and the scalability of the implementation. In the simplest setting, clients may connect to a single Siena server. However, servers can also dynamically connect to other servers to form a networked implementation. Two implementations of Siena are currently available, one using a hierarchical network of servers, and the other using an acyclic network. Interconnected Siena servers maintain dynamic routing information for publish/subscribe data by propagating subscriptions to other servers, as necessary.

Implementations of Siena are available for general download and use.[8]

2.8 STEAM

STEAM [16], developed at Trinity College of Dublin, is a publish/subscribe system specifically designed to be deployed over *ad hoc* networks. Because of the volatility of *ad hoc* network connections, the design of STEAM cannot rely on a single component providing a system-wide service. STEAM is therefore inherently distributed and allows consumers to subscribe by contacting a local network service interface [17]. STEAM

[2] http://www.jboss.org/
[3] http://openjms.sourceforge.net
[4] http://www.objectweb.org/joram/
[5] http://www.fiorano.com/products/fmq/
[6] http://openorb.sourceforge.net
[7] http://www.research.att.com/~ready/omniNotify/
[8] http://www.cs.colorado.edu/serl/siena/

implements *subject* and *content* subscription filters. Also, since it is designed for mobile domains, it implements a *proximity* filter. Subject and proximity filters are applied on the producer side, while other filters are applied on the consumer side.

Because if its related design goals, STEAM would be a good target for our mobility service evaluation. However, an implementation of STEAM is not available.

3 The Mobility Service

The general functional goal for our mobility service is to support the movement of clients between access points of a distributed publish/subscribe system. Our service does this by managing subscriptions and publications on behalf of a client application, both while the client is disconnected and during the switch-over phase. The service attempts to minimize duplication and, most importantly, loss of information. Our non-functional goals are portability to different publish/subscribe platforms, and adaptability to the characteristics of the given publish/subscribe system implementation, and to the characteristics of the underlying communication network.

Our design makes a few, relatively weak assumptions about the target publish/subscribe system. We assume a service API consisting of a general *subscribe* function and a general *publish* function. Notice that the subscribe function is stateful, meaning that it leaves behind state and modifies the behavior of the system. We are limiting the service to managing subscriptions only, and therefore we are assuming that mobile publishers will themselves manage outgoing messages before, during, and after migration. This can be easily done using a simple local buffer in the client. We do not make any assumptions about the data model and filtering language of the publish/subscribe system, but we do require that subscriptions and publications can be "serialized" for storage.

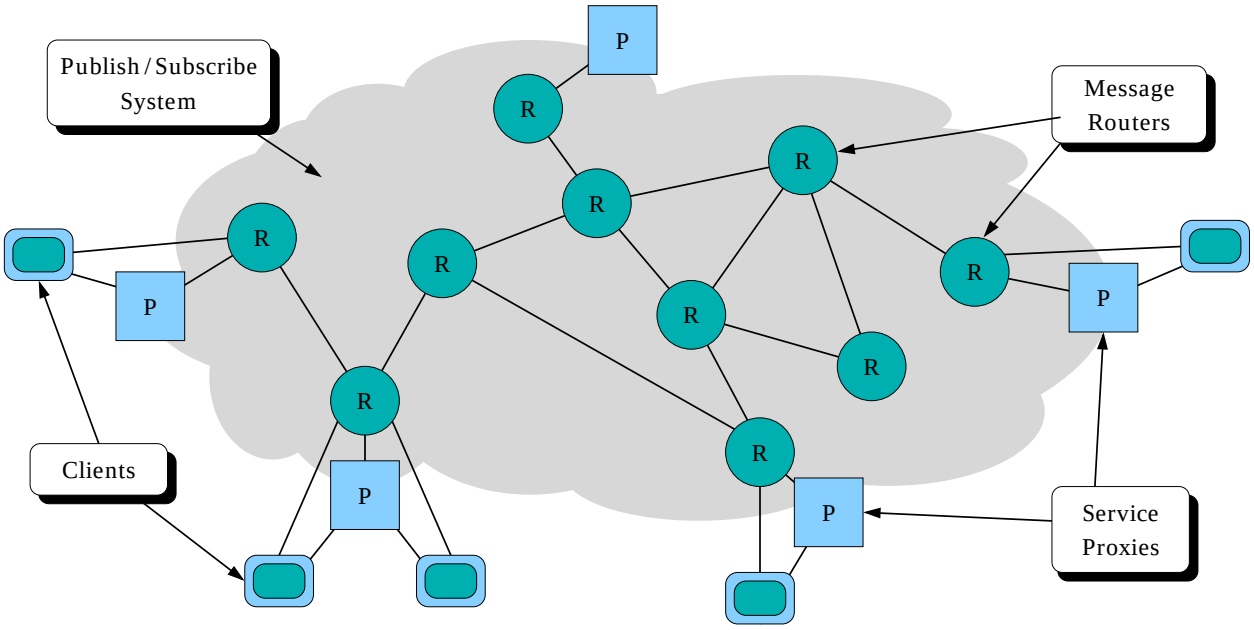

Figure 2: Mobility Proxy Objects

The mobility service is implemented by *mobility proxy objects*. Mobility proxies are independent, stationary components that run at the access points of the publish/subscribe system (see Figure 2). In addition to the proxy, mobile clients use a *mobility service library*, linked with the client application. The library wraps the target publish/subscribe API, mediating some of the requests made to the publish/subscribe system, and interacting with the mobility proxies during the move-out and move-in functions. Notice that while the proxy is largely independent from the target publish/subscribe system, the mobility library is tightly coupled

with the target publish/subscribe API. Whenever the language binding permits, the mobility library should be implemented as a derivative of the original target API by adding the move-out and move-in functions, and by overriding the subscribe function.

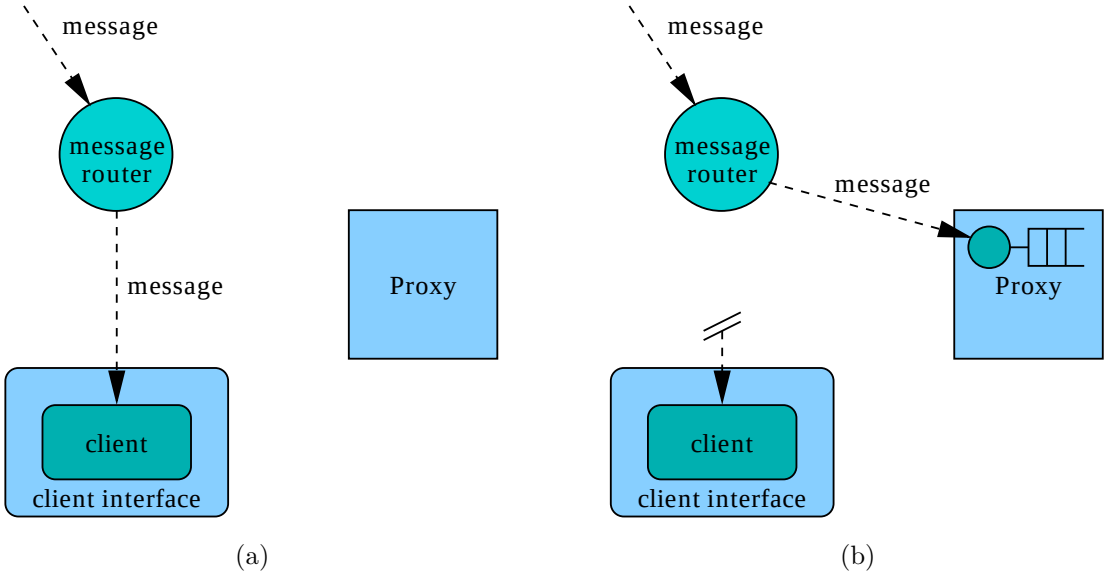

Figure 3: Connected and Disconnected Client States

The basic operation of the mobility service is quite intuitive. During normal (connected) operations, the client publishes and receives messages directly to and from the publish/subscribe system. However, while the the client is connected, the subscribe operation is mediated by the mobility library, which maintains a local copy of the client's subscriptions. Before detaching, the client calls the move-out function on its local mobility interface. The move-out function causes the local mobility library to transfer its stored subscriptions to its mobility proxy. The proxy proceeds to subscribe using all the client's subscriptions, and to store all the incoming messages in a dedicated buffer. Figure 3a shows a connected client, while Figure 3b shows a disconnected client. When the client reaches its destination, it uses the move-in function to instruct its mobility interface to contact a local proxy (termed a "move-in proxy"), passing it the address of the remote proxy from which it detached (termed a "move-out" proxy). At this point, the local move-in proxy and the remote move-out proxy engage in a protocol that results in the transfer of all the subscriptions and all the buffered messages from the remote site to the local site, and then onto the client library.

Notice that the client itself does not need to be aware of the move-out/move-in procedure. In fact, some monitoring process running on the mobile host, such as a power management daemon or a network connectivity daemon, might trigger both the move-out and move-in functions on behalf of all client applications running on that host. The same monitor could also take care of configuring the mobility libraries by discovering the address of a mobility proxy in the vicinity of the mobile host.

In the following sections we detail the architecture of the proxy and the synchronization operations of the mobility service.

3.1 Mobility Service Proxy

The mobility service proxy is a fixed component running on various sites throughout the publish/subscribe network. One can think of mobility service proxies as a local service station for mobile clients. A mobility service proxy can serve multiple mobile clients. Here, however, we only discuss the proxy from the viewpoint of an individual client. The mobility proxies of a distributed publish/subscribe system are independent from each other and do not form permanent connections. The only connections between proxies are established as a consequence of a move-in function.

The internal architecture of a mobility service proxy is illustrated in Figure 4. The proxy consists of

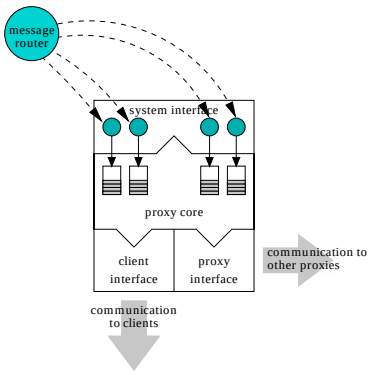

Figure 4: Internal Architecture of A Proxy

a *proxy core*, which implements the core functionalities of the proxy, including the buffering for received messages and the synchronization services, plus three interface components: the *client interface*, the *proxy interface*, and the *service interface*, which manage the interaction with the client, with the other proxy (move-in proxy or move-out proxy, as the case may be), and with the publish/subscribe system, respectively.

3.2 Basic Mobility Services

When a client invokes the move-out function, the client's mobility interface invokes the move-out function on the mobility proxy, passing a unique client identifier, a list of subscriptions, and an optional quality-of-service specification (discussed in Section 3.3). The mobility proxy executes the move-out function by creating a client proxy object with the given identifier, and subscribing that client proxy with all the subscriptions passed by the client's library. When the proxy is finished subscribing, it immediately sends an acknowledgment back to the client library. Upon receiving the acknowledgment from the proxy, the client library detaches the client from the publish/subscribe system and returns from the move-out function. The library may choose to detach the client either by suspending its connection or by unsubscribing the client completely, depending on whether or not the service supports a suspend function.

When a client invokes the move-in function, the client library may reconnect to the same proxy, which represents a situation in which the client has not moved, but was simply disconnected. In this case, the library invokes an abbreviated move-in function that does not involve the transfer of messages from one proxy to the another. The client library just restores the client's subscriptions, either by resuming the previously suspended connection or by reissuing all the client's subscriptions, and absorbs the buffered messages. In particular, using the unique client identifier passed as a parameter to the move-in function, the proxy retrieves the proxy object associated with that identifier and transfers to the client library all the messages buffered by the proxy object. Finally, the proxy unsubscribes and destroys the proxy object.

In the situation where the client does indeed move from one access point to another, the client library will connect to a different, remote proxy and invoke the full move-in function. Of course, the full move-in function is a bit more complex. It uses the unique client identifier, the address of the move-out proxy, and an optional quality-of-service specification as parameters, and proceeds in the following steps illustrated in Figure 5.

1. The client library activates a local merge queue and starts up a receiver process to manage the queue;

2. The client library restores all the client's subscriptions by subscribing the receiver process at the new access point. At this time, some messages matching the client's subscriptions may be delivered directly by the router to the receiver process, which stores them in the merge queue;

3. The client library sends a *message download request* to the move-in proxy, specifying the address of the move-out proxy and the client identifier.

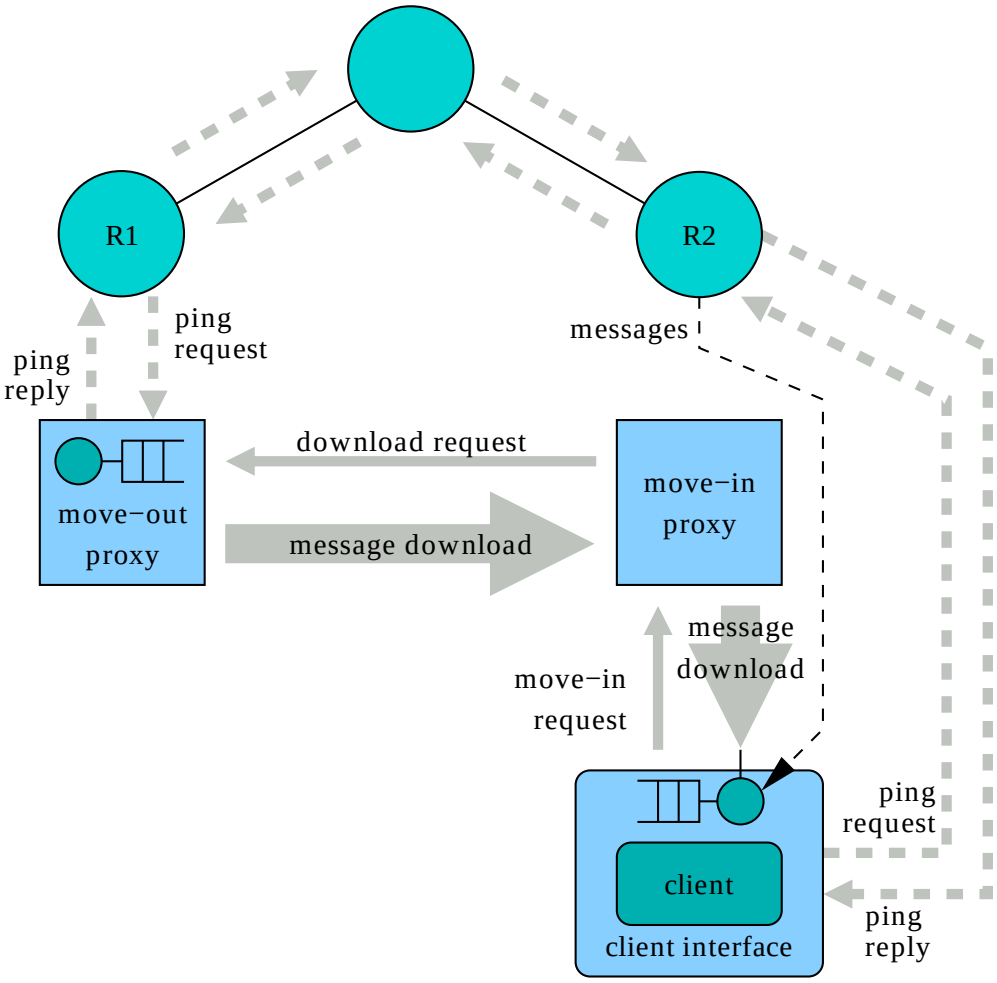

Figure 5: Full Move-In Function

4. The move-in proxy contacts the move-out proxy and downloads all the messages buffered by the client proxy associated with the given identifier. The move-out proxy then unsubscribes its client proxy object, sends all the buffered messages to the move-in proxy, and destroys the client proxy object. The move-in proxy forwards every downloaded message to the client library, which stores them in the merge queue.

5. After the client library has received the last message downloaded from the move-out proxy, the client library redirects the client's subscriptions from the receiver process to the actual client, and destroys the receiver process.

6. Finally, the client library merges the messages from the merge queue, passes them to the client, and destroys the merge queue.

Notice that some publish/subscribe systems provide a switch function that can be used to redirect messages from one contact point to another. In cases in which this feature is absent, two main strategies are available to the client library when switching from the receiver process to the actual client. In one case, the client library may choose to implement the receiver process as a wrapper to the client, and therefore to always have the wrapper intercept received messages and pass them over to the actual client. In the other case, the client library would have to implement another local, and therefore simpler, switch-over process by subscribing the actual client and unsubscribing the receiver process. Both solutions have some clear advantages and disadvantages.

3.3 Synchronization Options

It is easy to see that both the move-out and move-in functions are characterized by some phases in which messages may be lost or duplicated. Specifically, messages may be duplicated in the following phases:

- during the move-out function, after the proxy has subscribed its local client proxy, and before the actual client has detached, since the same message may be delivered to both the client and the client proxy; and

- during the move-in function, after the client library has set up the receiver process, and before the move-out proxy detaches its client proxy, since the same event may be delivered to both the client proxy and the receiver process.

Messages may be lost in the following phases:

- during the move-out function, after the actual client has detached and before the subscriptions of the client proxy become effective; and

- during the move-in function, after the move-out proxy detaches its client proxy, and before the subscriptions of the receiver process become effective.

The black-out periods may or may not occur, depending on the semantics of the subscribe function implemented by the target publish/subscribe system. For example, in some publish/subscribe systems, subscriptions are activated by established "routing" paths using subscription information (e.g., in Siena [6, 7]), and therefore the implementation may return from the subscribe function before all the necessary routing information is in place and stable. This issue is discussed further in the next section.

To avoid duplications, the mobility service may attempt to exclude duplicates from the merge queue. However, this idea has two major limitations. The first problem is that the merge algorithm would require a comparison operator for messages objects that the publish/subscribe system may not provide and whose semantics may not be completely clear. The second, and probably more important problem is that a merge algorithm would also require that messages be somehow unambiguously identified so that the merge operation could distinguish a pair of syntactically identical, but distinct messages, from a pair of messages that are copies of the same original message. For these reasons we have decided not to remove duplicates in our merge operation.

Avoiding losses is, to a large extent, possible and our mobility service provides two classes of mechanisms. They are applied to both the move-out and move-in functions, and are designed to assure some synchronization between the publish/subscribe system and the mobility service so as to avoid black-out periods. In particular, when switching a set of subscriptions from point A to point B, the main idea behind these mechanisms is to make sure that the subscriptions at B are active before proceeding to remove the subscriptions at A.

The fist mechanism is based on a "ping" message sent from B to A and back using the publish/subscribe system itself. For example, during the move-out function, move-out proxy A will subscribe for a "ping request from B". The client library B will then subscribe for a "ping response from A" and will start publishing a series of "ping request from B" messages at regular intervals. Upon receiving a ping message from B, the move-out proxy A will publish a set of ping responses, one of which will eventually reach B. It is only after receiving the response from A that the client library issues the download request that will eventually cause A to remove its subscriptions. This exchange is illustrated in Figure 5.

The effectiveness of the ping message relies on the general idea that B having received A's ping response implies that the necessary routing information for that message is in place between A and B, which is a good indication that the necessary routing information is also in place for every other subscription (previously) issued by B from any other publisher X to B. Notice that the ping receipt does not guarantee that the necessary routing information is in place in all cases. In fact, in a publish/subscribe system based on a generic interconnection topology, the routing information may well reach A before it reaches X. However,

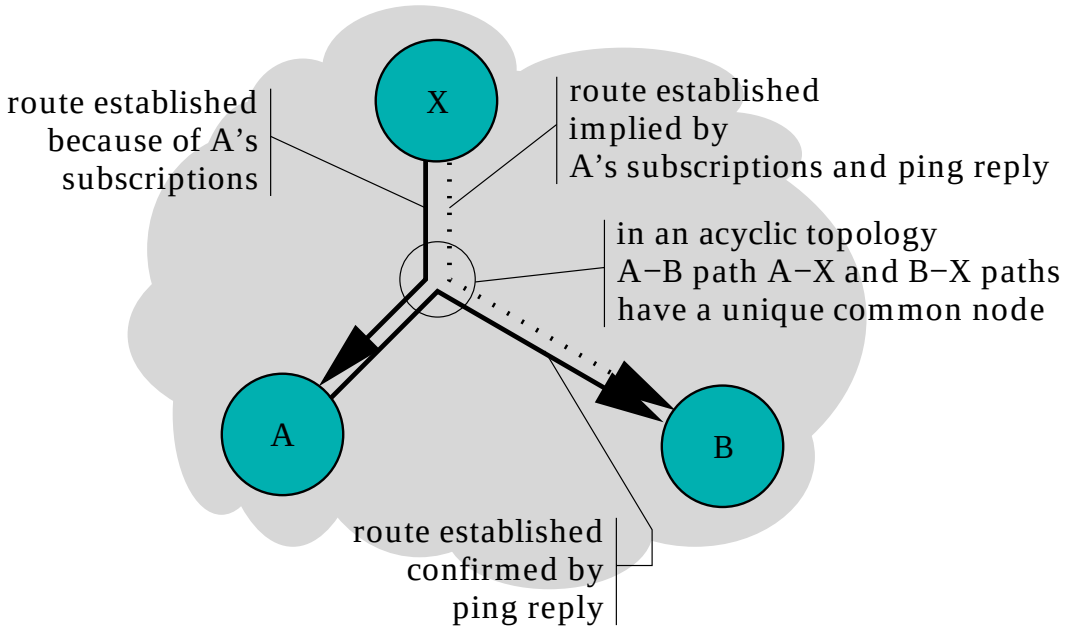

Figure 6: Implication of a Ping Reply with an Acyclic Topology

in an acyclic topology, which is the only topology available to us today, the routing information propagating from B will certainly reach some point between B and A (possibly A, B, or X) that is already connected to X, since the same set of subscriptions are active between X and A. This concept is illustrated in Figure 6.

The second mechanism consists of a simple delay. The idea here is not to bother with the setup and execution of the ping protocol, but instead to simply wait for some configurable amount of time before removing the subscriptions from point A. Besides its simplicity, this method has the additional benefit of not adding traffic to a possibly congested system. The obvious disadvantage of this mechanism is that, unlike the ping protocol, it is not adaptive, and therefore can only provide a limited and variable assurance of consistency.

The relative performance of these mechanisms is evaluated in the next section.

4 Evaluation

The objective of our work is to create a value-added, portable, and adaptive mobility support service for the clients of a distributed publish/subscribe system running on mobile, wireless devices. This objective defines, more or less explicitly, a framework for evaluating the work. In particular, the goals of the evaluation study are the following.

- *To assess the benefits of using the mobility support service.* To satisfy this goal, we ask ourselves how well a client application would perform under various workloads with and without the support service.
- *To assess the portability of the architecture and service implementation.* In this case, the question we ask is how easy it is to re-target the implementation of the service to a new publish/subscribe system.
- *To assess the adaptability of the service and, particularly, its synchronization mechanisms.* This evaluation goal leads to a comparative analysis of the performance of the synchronization mechanisms across a valid sample of publish/subscribe implementation features.
- *To confirm the validity of the service design on the network substrates used by target applications.* This demands a performance evaluation across heterogeneous networks, ranging from fast and reliable wireline LANs to much slower and less reliable wireless links.

In order to answer the questions listed above, we implemented the mobility support service on top of three distributed publish/subscribe systems, and then experimented with the different implementations using a simple mobile client application under various workloads and network configurations. For each experiment, we recorded a log of events, including every publication, every received message, and the client's movement operations. From these logs we were able to collect aggregate metrics, such as the number of lost or duplicate messages and any delay resulting from a move-in function. We also plotted the logs in a style of graph that gives visual clues for many of the metrics.

The remainder of this section details the experimental setup as well as the most prominent results of our analysis.

4.1 Experimental Setup

We implemented the mobility support service for Elvin, FioranoMQ, and Siena. Details of these publish/subscribe systems are available in sections 2.1, 2.5.1, and 2.7, respectively. Although this is a somewhat limited set of target systems, we believe that they represent a valid sampling of features and interface types, tempered by what was available to us for experimentation.

In our experiments, we used a fixed publish/subscribe overlay network of three nodes, a single stationary publisher client, and a single mobile subscriber client. Again, this is a limited configuration, but still rich enough to illuminate significant differences among the test subjects. We mapped this configuration

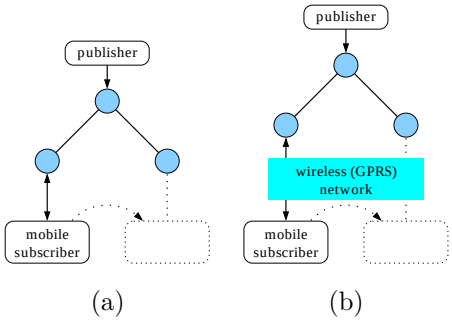

Figure 7: Experimental Network Setup

onto two types of network infrastructures. In both mappings, the whole publish/subscribe system and the publisher node reside on a wireline, low-latency, high-bandwidth network. In the first configuration

(Figure 7a), the mobile client also connects to the publish/subscribe system through wireline links, while in the second configuration (Figure 7b), the mobile client connects to the publish/subscribe system through a simulated GPRS network. Specifically, we set up the first configuration using four computers running the publish/subscribe system and its clients, connected using a LAN. (The publisher client runs on the same host with its publish/subscribe server.) For the second configuration, we used the same four computers, with three of them connected over a LAN. The fourth runs the mobile client application connected through an IP network tunneled through a wireless network simulator called Seawind [14, 15].

Base-Rate	13400 bps (uplink/downlink)	Capacity of an individual channel
Max-Rate	2 uplink, 3 downlink	Maximum number of channels per user
Available-Rate	1–2 uplink, 1–3 downlink	Available channels
Error-Type	UNIT	Error probability applies to packets
Error-Probability	10^{-3} (uplink/downlink)	Error probability
Error-Handling	DELAY_ITERATE (uplink/downlink)	Link-level error-handling (GPRS network retransmits broken packets)
Error-Delay-function	30–50ms (uplink/downlink)	Delay used in retransmission function

Table 2: Seawind GPRS Parameters

Seawind is a generic wireless network simulator parameterized to reflect the characteristics of various types of wireless connections. In our case, Seawind works by emulating a point-to-point communication channel extending over a GPRS network. One end of the channel represents the mobile device, running the mobile publish/subscribe client application, while the other end represents the remote host, running a publish/subscribe access point. In practice, Seawind creates two PPP [20] network protocol adapters, one for the mobile-device side and one for the remote-host side, which we configure in such a way that all IP traffic between the mobile client and the publish/subscribe access point will be routed through those adapters and, therefore, through Seawind. In processing through-traffic, Seawind emulates the behavior of a GPRS network according to its configuration parameters, thereby introducing characteristic delays, errors, and packet loss. The parameters we used in our experiments are listed in Table 2 and correspond to a class-6 mobile station with two and three timeslots in uplink and downlink, respectively, over a GPRS network with coding scheme CS-2 [1].

s	number of distinct subscriptions	$1, 2, 5, 10, 20, 50, 100$
p	publication rate (messages per second)	$1, 2, 5, 10$
Δ	migration interval (seconds)	$10, 30$

Table 3: Experimental Publish/Subscribe Parameters

In all our experiments, the subscriber issues s subscriptions, while the publisher publishes a series of publications at a constant rate of p publications per second. Subscriptions and publications are such that every publication matches all subscriptions. Although this might seem an extreme and unrealistic case for a single publisher and a single subscriber, it fits our goal of testing the mobility support service under a steady flow of messages. (Notice, in fact, that evaluating the publish/subscribe system itself would require a much richer distribution of subscriptions and publications. However, such experiments are outside the scope of this paper.) Every experiment presented here consists of a single migration of the mobile subscriber client, for a fixed disconnected interval of $\Delta = 10$ seconds. This, too, is perhaps an extreme case, corresponding for example to a person moving around a building, using a PDA connected to a highly distributed publish/subscribe system through a multi-point wireless network.

As it turns out, it is the combination of high numbers of subscriptions (s) plus the total number of messages buffered during a disconnection period ($\Delta \times p$) that stress tests the mobility support service. So, in that respect, our chosen parameters generate a fairly heavy load on the mobility support service.

4.2 Overall Overhead and End-to-End Benefit

Before we explore the detailed results of our experiments, we briefly review the overall impact of adding the mobility support service, both in terms of performance overhead and in terms of end-to-end delivery benefit during a client's movement. This brief presentation also introduces the main type of graph we use to display the behavior of a mobile publish/subscribe application.

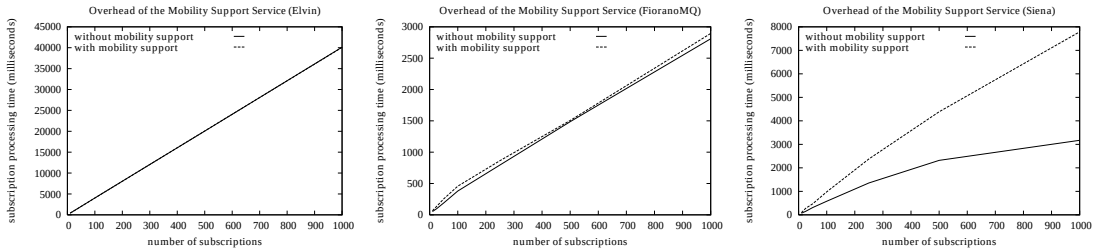

Figure 8: Overhead for a Stationary Client

Figure 8 shows the overhead incurred by the mobility support service when used with Elvin, FioranoMQ, and Siena. The graphs show, for a stationary client, the total subscription processing time as a function of the number of subscriptions. This provides a baseline indication of the overhead. With Elvin and FioranoMQ, the mobility service incurs a barely noticeable overhead. With Siena, it incurs a more significant overhead.

The reason for this difference in overhead lies in the semantics of subscriptions adopted by the three systems. In particular, both Elvin and FioranoMQ treat all subscriptions from the same client application as separate entities, associated one-to-one with the object designated to process the messages matching the subscription. This choice of semantics allows the our client library to intercept and immediately store all subscriptions in a table indexed by the identifier of the associated processor object, where a new subscription for an existing processor object will simply overwrite the old one. While this choice simplifies our job, it also introduces practical limitations for client applications and inefficiencies for the system implementation. In fact, the one-to-one association forces applications to create and manage one processor object for each individual subscriptions, and also leads to implementations that deliver a message matching multiple subscriptions in multiple copies to multiple processor objects within the same application.

Siena, on the other hand, eliminates those limitations and inefficiencies by adopting a stateful subscription semantics based on the concept of *covering relations* [5]. The details of covering relations are beyond the scope of this paper, but briefly, the covering relation is used to decide which new subscriptions to save and which ones to ignore based on their relation to all the other subscriptions that have been previously seen. As each new subscription arrives, processing is performed to determine whether there is a previous subscription more general than the new subscription and, if so, the new subscription is ignored. The purpose for doing this in Siena is to minimize the amount of traffic and processing given over to setting up routing and forwarding information in the nodes of the overlay network. The mobility service interface implemented for Siena duplicates this processing (using functions conveniently available in the Siena API to perform these checks) in order to minimize the set of subscriptions that must be stored by the client library. Of course, one could instead design an opportunistic simplification process that would run intermittently to eliminate redundant subscriptions. While we have not designed such a mechanism, we would expect it to perform somewhat better than the current implementation.

Figure 9 shows a first result indicating the general benefit of the mobility support service—that is, to guarantee an uninterrupted flow of messages, as seen from the viewpoint of a client application. More detailed results are given in Section 4.3, below. The top two graphs show the case of a mobile client executing without the benefit of mobility support, while the bottom two graphs show the same client using the mobility support service. The left-side graphs show results in a wireline LAN, while the right-side graphs show results for the simulated wireless GPRS network. All four graphs refer to the Siena implementation in a scenario of 50 subscriptions and a publication rate of two publications per second.

Let us explain how the graphs in Figure 9 display the results. The figure shows four graphs that we call *departure/arrival traces*. The departure time is when a message is published, while the arrival time is when

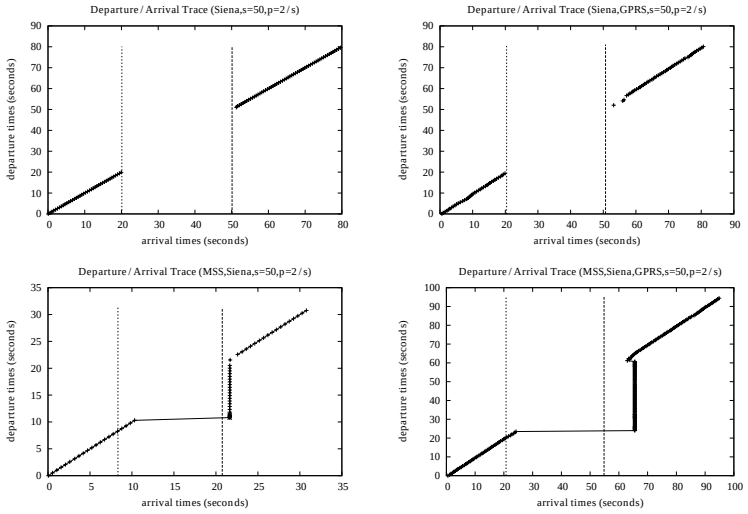

Figure 9: End-to-End Effect of Mobility Support

the message is actually received by the subscriber. Every point corresponds to a message received by the mobile client. Points are connected by lines representing the sequential ordering of messages as they are published by the publisher. Missing segments correspond to messages published by the publisher, but never delivered to the mobile subscriber. The two vertical dashed lines represent, respectively, the time when the subscriber invokes the move-out function and the time when the subscriber invokes the move-in function.

In the case without mobility support, the vertical dashed lines simply represent the times when the client detaches from one access point and reattaches to the new one. Notice in the top two graphs of Figure 9 that all messages during this interval are lost. In the case of mobility support, the vertical dashed lines represent not only the detach/reattach times, but also when the major processing provided by the service begins, especially message buffering and message merging. Notice in the bottom two graphs of Figure 9 that nearly all published messages eventually arrive at the client, but only after some amount of delay caused by the processing associated with the move-in function. Furthermore, notice that there can be a delay between when the client requests to be detached and when the client actually stops receiving messages. This delay is due to the processing associated with the move-out function.

The departure/arrival trace gives a good visual sense for the behavior of the mobile publish/subscribe application. Below we use this representation to display more detailed results concerning the mobility service.

4.3 Wireline Network

The next set of experiments are based on a wireline network configuration. We show a series of results for the mobile application, first without any mobility support, then with the basic mobility support service, and finally combining the basic service with various synchronization mechanisms.

Figure 10 shows the departure/arrival traces of the mobile client for the Elvin, FioranoMQ, and Siena implementations, but without using our mobility support service. As mentioned above, the messages published during the disconnection period are lost. Notice also that with Elvin some messages are also lost during a short transition period immediately after the mobile client reattaches (i.e., after the move-in function is performed). This behavior is due to the fact that the client library reissues all the subscriptions (100 of them in this experiment) upon reattaching; during this period of time, the publish/subscribe system and the underlying network infrastructure are busy handling the subscriptions and dropping published messages. We did not notice this behavior in experiments with fewer subscriptions.

The traces shown in Figure 11 represent the behavior of the application using the basic mobility service implemented on top of Elvin, FioranoMQ, and Siena. The nearly horizontal line indicates the period of time during which the mobile client is not receiving any messages. Specifically, it is the interval between the last received message after the move-out function is invoked and the first received message after the move-in

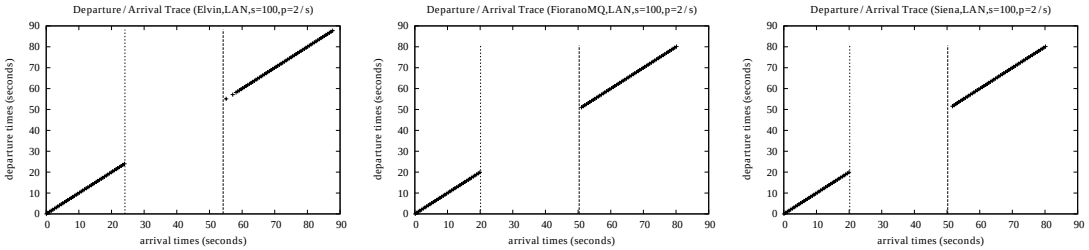

Figure 10: No Mobility Support

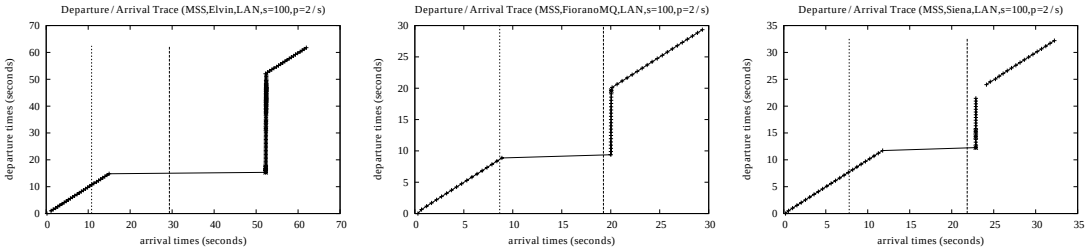

Figure 11: Basic Mobility Support with Many Subscriptions

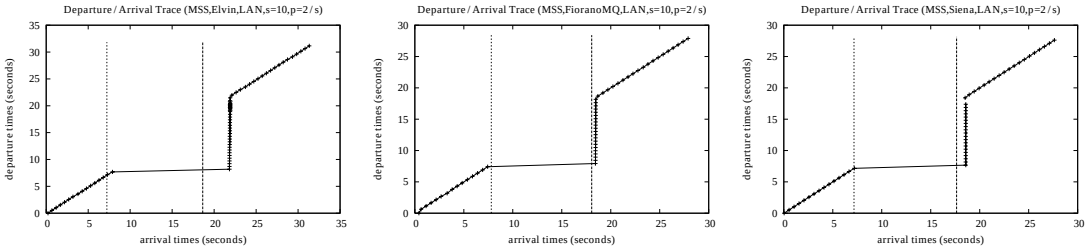

Figure 12: Basic Mobility Support with a Few Subscriptions

function is invoked. The almost vertical sequence of message arrivals, which occurs some time after the client reattaches, indicates the delivery of the messages buffered by the mobility support service.

The graphs highlight two important quantities. The first is the interval between the detach time and the time the last message is received before the client becomes completely disconnected. This delay represents the move-out processing time, which is essentially the time it takes for the mobility proxy to subscribe on behalf of the client. The second quantity is the corresponding delay after the reattach time, which represents the move-in processing time, and amounts to the duration of steps 2 and 4 of the move-in procedure presented in Section 3.2). In our experiments, with 100 subscriptions, the service shows better performance on FioranoMQ, than on the other two systems. On both Elvin and Siena, the application incurs a significant delay after the move-out function is invoked. Moreover, as mentioned above, on Elvin the service also suffers a performance degradation with the move-in function. Again, these delays are due to the fact that the system is busy processing subscriptions.

We also ran the experiments using only 10 subscriptions. The results, shown in Figure 12, indicate much better performance for the Elvin and Siena implementations, with only minimal move-out and move-in delays.

Another noticeable phenomenon evident in the graphs of Figure 12 and Figure 11 is how the Siena implementation loses some message after the move-in function. As explained in Section 3.3, messages are lost because the publish/subscribe system may return from the subscribe operation before all the necessary routing information has been propagated. Siena is the only system of the three that suffers from this problem because it is the only one that employs a truly distributed message routing algorithm. While the propagation of subscriptions is a prominent and documented feature of Siena, neither Elvin nor FioranoMQ specify how they treat subscriptions (and publications) when used in a distributed federation. Specific experiments conducted using simple network monitors indicate to us that they maintain subscriptions local to their access point, and that they broadcast publications among those access points. Thus, the designers of FioranoMQ and Elvin evidently chose to incur the cost of flooding the network with publications, rather than trying to optimize message traffic through a subscription-based routing protocol. This approach shows benefits when a mobile client reattaches, but is quite costly at all other times.

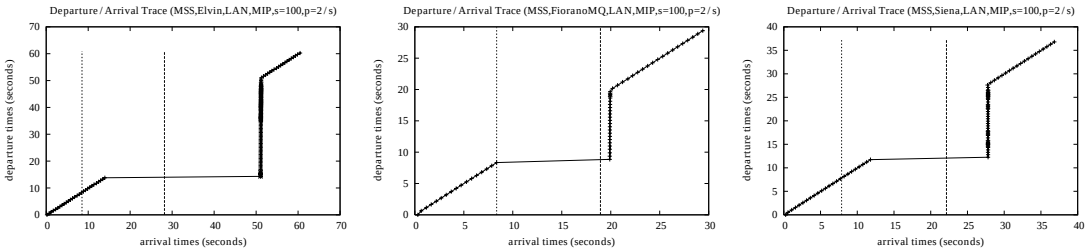

Figure 13: Mobility Support with Ping Option upon Reattach (Compare with Figure 11)

The precise purpose of the optional synchronization mechanisms described in Section 3.3 is to reduce messages losses during the move-out and move-in functions. Therefore, we have run all the previous experiments with the synchronization options turned on. Figure 13 shows the traces for Elvin, FioranoMQ, and Siena, running the optional ping during the move-in function, for the same scenarios of Figure 11. A comparison of these two series of graphs confirms that the ping option is an effective synchronization mechanism. In fact, we notice that Elvin and FioranoMQ have essentially the same behavior with or without the ping option, incurring the same delay. On the other hand, Siena incurs a slightly greater delay with the ping option. This delay is the price paid for the reduction of losses. The increase in reliability is evidenced by the fact that the departure/arrival trace of the experiment with ping does not show holes, which are instead visible in the traces of the same experiment without ping.

In practice, this means that the ping option is effective in reducing lost messages caused by a delay in propagating subscriptions, as in Siena. Moreover, this is achieved without penalizing implementations that do not experience such delays because they keep subscriptions local and broadcast their messages, as in Elvin and FioranoMQ.

This fundamental observation about the ping options leads to two further questions. How does the simpler

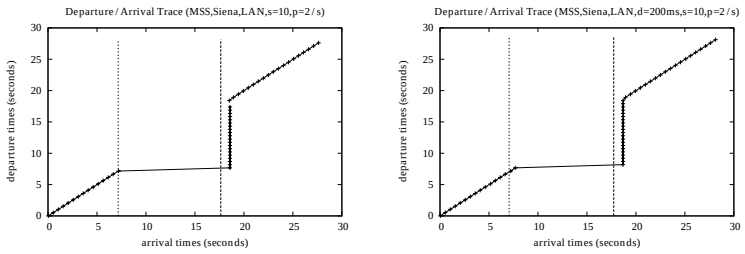

Figure 14: Benefits of the Delay Option

delay option compare with the ping option? Also, since we can hypothesize an inverse correlation between the number of lost messages and the number of duplicate messages, how does the ping option affect the number of duplicate messages? The two graphs of Figure 14 provide an initial answer to the first question. The trace on the left side shows an experiment using the basic service in the presence of 10 subscriptions. This trace shows a message loss right after the move-in function. The trace on the right corresponds to the same setting of subscription and publication rate, but this time using the delay option with a delay of 200ms. The trace on the right shows no losses, which gives us a confirmation that the delay option is effective.

As it turns out, however, Figure 14 does not provide strong evidence that the delay option is effective in all cases. In fact, our experiments confirm the intuition that the delay that would be necessary to establish routing information is proportional to the number of subscriptions the client restores. Also, although we have no detailed evidence to confirm this, we can easily argue that the delay depends upon the characteristics of the underlying network.

In order to answer the second question, we need to look at this experimental data from a different viewpoint. In fact, although the departure/arrival traces give us some immediate visual clues, they do not show the precise count of duplicate messages, nor do they clearly show the precise amount of delay and the total number of lost messages. Fortunately, we were able to derive and plot these quantities from the logs collected from our experiments.

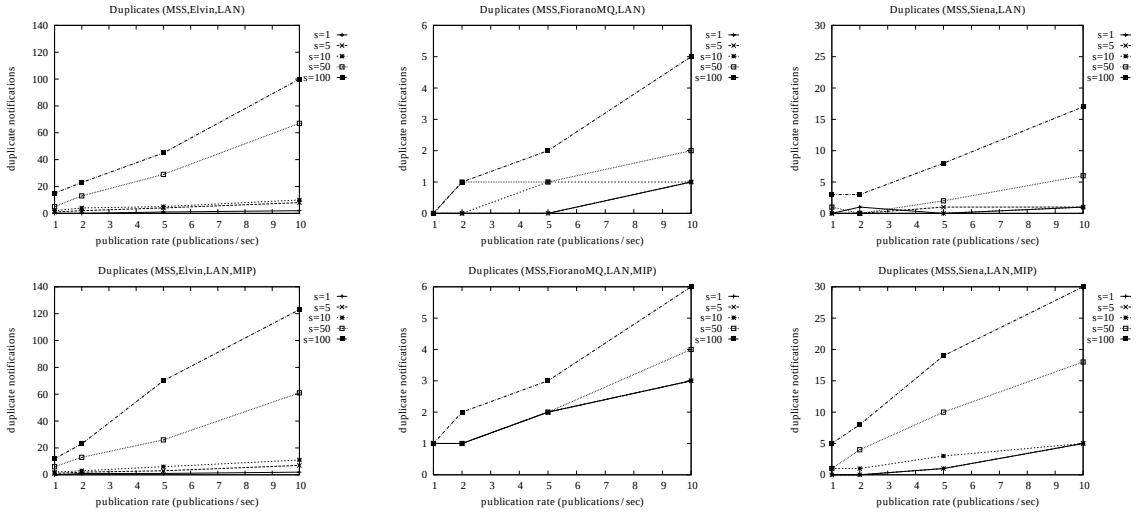

Figure 15: Duplicate and Lost Messages upon Reattach With and Without Ping

Figure 15 shows measurements of duplicate messages during the move-in function. These measurements are given as a function of the number of subscriptions and of the publication rate. The top row shows the basic mobility service, while the bottom row shows the experiments that use the ping option. For convenience in comparing the results pairwise, the graphs are set with the same vertical range for each system. These graphs give use two confirmations. First, once again, we note that the ping option has only a very limited cost

in terms of added duplicates on Elvin and FioranoMQ, which in fact do not need that option. Conversely, it increases the number of duplicates on Siena (in the worst case by 45%), which benefits from the ping option. The second and also intuitive confirmation is that the number of duplicates is proportional to the number of subscriptions and to the publication rate.

4.4 Wireless Network

In order to analyze the behavior of the mobility support service over a wireless network infrastructure, we conducted a second set of experiments, replicating all the combinations of publish/subscribe parameters and mobility support services explored in the wireline case. As mentioned above, the configuration for these experiments consists of a mixed network that includes a simulated GPRS link between the mobile client and the publish/subscribe system. Here, we report the interesting differences we found between the wireline and wireless configurations.

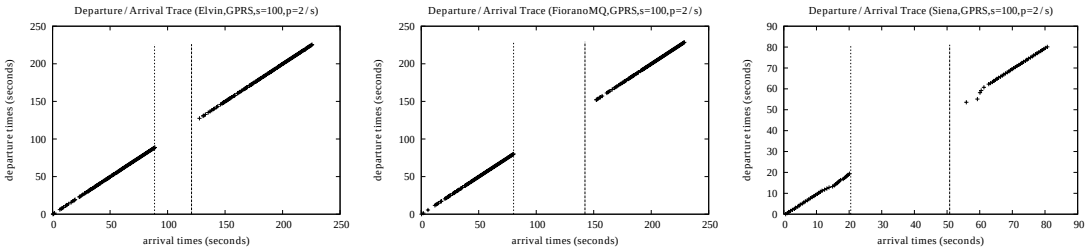

Figure 16: No Mobility Support (Wireless Infrastructure)

Figure 10 shows the departure/arrival traces of the mobile client over a simulated GPRS network without the mobility support service. Notice how these traces show remarkably different behaviors than the corresponding cases on the wireline network (see Figure 10.) The presence of the slow, unreliable link between the client and the publish/subscribe system amplifies the loss of messages after the client reattaches to the network. Notice, too, that a similar effect is visible at the beginning of the trace, when the client attaches to the network for the first time. Finally, notice the difference between Elvin and FioranoMQ on the one hand, and Siena on the other. Elvin and FioranoMQ are noticeably slower than Siena in this case. We have not performed additional experiments to analyze this difference. However, we believe that the reason for this difference is that Elvin and FioranoMQ adopt communication primitives that are less bandwidth efficient, and that favor reliability over speed, whereas Siena adopts a bandwidth-efficient "best effort" approach.

The presence of the GPRS link has an impact, as one would expect, on the use of the mobility support service. However, in two out of three cases (Elvin and Siena), the performance of the mobility support service is comparable to the corresponding cases on a wireline network. Figure 17 shows traces that use mobility support without options, with 100 subscriptions and 10 subscriptions. Notice that, comparing Figure 17 to Figure 11 and Figure 12, the only implementation that shows a noticeable performance degradation is FioranoMQ. Again, we believe that this difference is due to the particular choice of communication mechanism adopted for FioranoMQ.

As we did in the wireline case, we tested the effectiveness of the ping option in combination with the GPRS network. Once again, the experiments show that only Siena looses messages during the move-in function, and therefore that Siena is the only system that needs the ping option. Nonetheless, the experiments also confirm the adaptive behavior of the ping option that causes only minor performance degradation to Elvin and FioranoMQ.

Another conclusion we can draw by examining the data of Table 4 is that the mobility service suffers a significant slowdown on the move-in function on Elvin and FioranoMQ over GPRS. Although we have performed only a few experiments to investigate this behavior in detail, we believe that it is due to the choice of communication protocol. In our experiments, we determined that both Elvin and Fiorano use one-time TCP connections to exchange messages and subscriptions, while Siena uses *keep-alive connectors* that attempt to re-use the same TCP connection to pass multiple messages and/or subscriptions between clients and message routers (and between message routers.) As it turns out, this simple optimization of

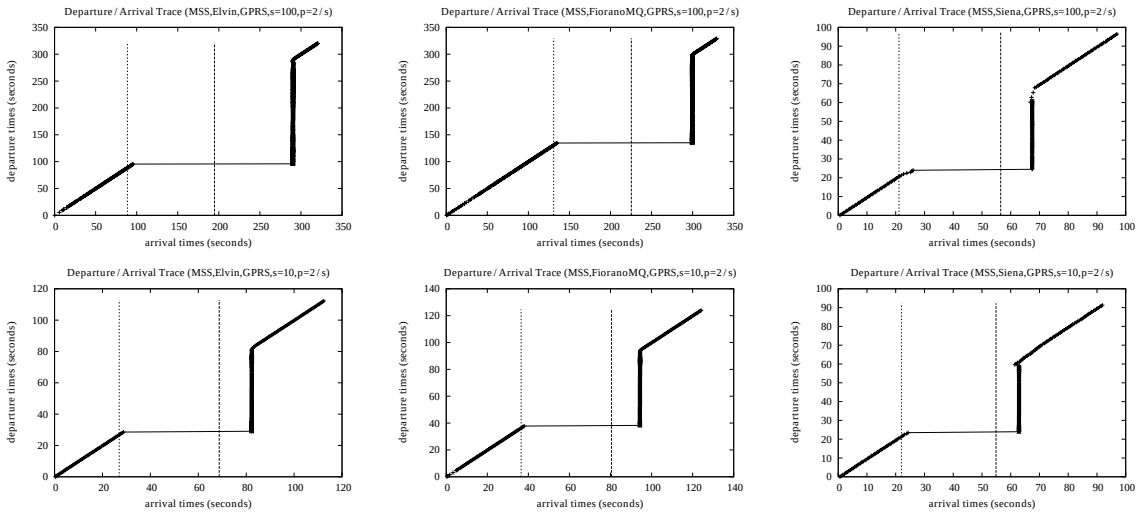

Figure 17: Basic Mobility Support with 100 and 10 Subscriptions

Elvin

s	delay		losses		duplicates	
	basic	ping	basic	ping	basic	ping
10	13551	17151	0	0	18	22
50	48608	52525	0	0	76	88
100	95031	98798	0	0	148	159

FioranoMQ

s	delay		losses		duplicates	
	basic	ping	basic	ping	basic	ping
10	13932	15747	0	0	11	18
50	40544	42215	0	0	43	50
100	74773	77304	0	0	90	90

Siena

s	delay		losses		duplicates	
	basic	ping	basic	ping	basic	ping
10	8294	11751	0	0	7	12
50	10984	23585	2	0	4	17
100	10841	39608	10	0	4	30

Table 4: Summary of Delay, Losses, and Duplicates upon Reattach over GPRS for $s = 10, 50, 100$ and $p = 2/sec$

the communication within the publish/subscribe system yields a significant performance improvement over slow and unreliable communication channels. We must insist that we do not intend to compare the three publish/subscribe systems we used. In fact, we should also say that Elvin too features an optional communication mechanism based on persistent connections, which we found out about only after analyzing these results.

4.5 Portability

Clearly, providing a comprehensive assessment of the portability of the mobility support service is a very difficult task. Nonetheless, our experience in successfully implementing the service on three, rather different publish/subscribe systems, and obtaining comparable, feature-independent behaviors, is an initial confirmation that the service is indeed portable.

core service	639
Elvin interface	797
FioranoMQ interface	936
Siena interface	755

Table 5: Mobility Support Service Implementation Sizes

We believe that the primary reason for the portability of the mobility support service is its simplicity. We do not have strong data to confirm our hypothesis, and a more thorough analysis is well beyond the scope of this paper. What we can show is a simple measurement (see Table 5) of lines of code for the core mobility service (proxy and library) and for its specializations for Elvin, FioranoMQ, and Siena.

5 Conclusions and Future Work

In this paper, we presented the architecture of a mobility support service for distributed publish/subscribe systems. The specific goal of this service is to provide an added-value support to the mobile clients of a publish/subscribe system. In simple terms, the support we offer consists in buffering incoming publications and in handling the switch-over of subscriptions from one publish/subscribe access point to another. With this service, we also implemented two synchronization options designed to reduce the loss of publications during the switch-over process. Both the basic service and its options are intended to be independent from the target publish/subscribe service. Yet they are also designed to adapt to the numerous implementation features present in different target publish/subscribe services.

In addition to the design of the service, we presented the results of an extensive evaluation of our service. For this evaluation, we implemented the mobility service architecture on three distributed publish/subscribe systems and we ran over 2000 experiments for each one of the three implementations, under a variety of workloads and network configurations. The experimental data we have collected confirmes the validity of the general service architecture as well as the adaptive nature of its optional synchronization devices.

This work is part of our study on publish/subscribe systems and content-based networking. In this larger context, it is our first step in considering the issue of mobility of clients. We can think of two main extension paths for this work. One direction is towards a service with more options and more configurability. An example of such a useful configuration feature would be a mechanism to selectively exclude subscriptions or publications from the mobility support. For example, an application might decide to be notified only of the 10 most recent buffered events. In another case, an application might want to make some subscriptions "local" to its current site, and therefore ignore them in the switch-over process. The other research direction derived from this work is concerned with a tighter integration with the inner routing mechanism that characterize truly distributed publish/subscribe services. Such a study would analyze and possibly optimize the process of adjusting the routing information within the publish/subscribe system, in response to, or in anticipation of clients migrations.

References

[1] 3rd Generation Partnership Project - Technical Specification Group. *Digital Cellular Telecommunications System (Phase 2+); General Packet Radio Service (GPRS) Service Description; Services and System Aspects; Stage 2*, Jan. 2002.

[2] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 262–272, Austin, Texas, May 1999.

[3] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a global event notification service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Elmau, German, May 2001.

[4] P. R. Calhoun and C. E. Perkins. Mobile ip network access identifier extension for ipv4. RFC 2794, Mar. 2000.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, OR, July 2000.

[6] 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, Aug. 2001.

[7] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, Arizona, Oct. 2001.

[8] *CORBAservices: Common Object Service Specification*, July 1998.

[9] *Notification Service*, Aug. 1999.

[10] G. Cugola, E. Di 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):827–850, sep 2001.

[11] P. Fenkam, E. Kirda, S. Dustdar, H. Gall, and G. Reif. Evaluation of a publish/subscribe system for collaborative and mobile working. In *Proceedings of the Eleventh IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 02)*, 2002.

[12] A. Fugetta, G. Picco, and G. Vigna. Understanding code mobility. *IEEE Transaction on Software Engineering*, 24(5), May 1998.

[13] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Second ACM international workshop on Data engineering for wireless and mobile access*, pages 27–34. ACM Press, 2001.

[14] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, and K. Raatikainen. Seawind: a wireless network emulator. University of Helsinky, Finland.

[15] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, and K. Raatikainen. *Seawind v3.0 User Manual*. University of Helsinky, Finland, September 2001.

[16] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS '02)*, Vienna, Austria, July 2002.

[17] R. Meier and V. Cahill. Taxonomy of distributed event-based programming systems. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS '02)*, Vienna, Austria, July 2002.

[18] P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS '02)*, pages 611–618, Vienna, Austria, June 2002.

[19] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, pages 243–255, Brisbane, Australia, Sept. 3–5 1997.

[20] W. A. Simpson. The point-to-point protocol (ppp). RFC 1661, July 1994.

[21] Sun Microsystems, Inc., Mountain View, California. *Java Distributed Event Specification*, 1998.

[22] Sun Microsystems, Inc., Mountain View, California. *Java Message Service*, Nov. 1999.

[23] TIBCO Inc., Palo Alto, CA. *TIBR+: a WAN Router for Global Data Distribution*, 1996.

[24] D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communication of the ACM*, pages 92–102, 1999.