Using the Publish-Subscribe Communication Genre for Mobile Agents

Amir Padovitz, Seng Wai Loke, and Arkady Zaslavsky

School of Computer Science and Software Engineering, Monash University,
Caulfield East, VIC 3145, Australia
padovitz@bigpond.com, swloke@csse.monash.edu.au,
a.zaslavsky@monash.edu.au

Abstract. We advocate the event-based communication genre for mobile agent communities, which is useful for exchanging and disseminating large volumes of small lightweight messages. We propose that the publish-subscribe model complements the proprietary or standard agent communication languages. We describe how we implemented the event notification mechanism for mobile agents, and analyse experiments that demonstrate the Elvin-based event notification mechanism for communication between heterogeneous agents, in particular, Grasshopper agents and Aglets. We also discuss experiments for measuring message losses due to agent migration.

1 Introduction

Various Internet-scale applications have found mobile agents advantageous [2]. The problem of agent communication at run-time is very much a challenge, and communication between heterogeneous agents in multi-agent systems or, in other words, agents from diverse platforms or environments poses even a bigger challenge [11].

A different paradigm, which has been useful for asynchronous communication between loosely coupled components of distributed systems, is publish-subscribe event notification mechanism. In this paper, we investigate the use of Elvin, a typical stateless content-based event communication system [5] in a dynamic mobile agent environment. While implementations of publish-subscribe models may differ, the fundamental principles remain the same, and in this paper we use the name Elvin both as the name of a specific implementation and more significantly as a generic representative of the class of event notification systems.

We highlight below three benefits of event-based mechanisms in dynamic mobile agent environments:

- The concept of an event is a natural abstraction for the activities within the environment and the interactions among places and agents. We see the event mechanism as a communication genre complementary to other means of inter-agent communication. For transfer of large data chunks between agents or more structured conversations, other inter-agent communication mechanisms (e.g., FIPA

M. Schillo et al. (Eds.): MATES 2003, LNAI 2831, pp. 180–191, 2003.
© Springer-Verlag Berlin Heidelberg 2003

- ACL, www.fipa.org) can be used. The event service only serves as a means of communicating (potentially large volumes of) short messages among agents or to agents about events, in a lightweight manner.
- Event-based models provide implicit invocation [1]: a component A can invoke another component B without A being required to know B's *name* or B's *location*. Components such as B 'register' interest in particular 'events' that components such as A 'announce'. When A announces such an event, the event system notifies (or invokes) B, even though A doesn't know that B or any other components are registered. This suits a dynamic environment, where A need not know B's location since B might be continually moving, and A can interact with other components in the environment without knowing their details (e.g., their name, or how many agents there are, thereby providing scalability). Elvin's mechanism to achieve implicit invocation is *content-based addressing* (or *undirected addressing*) where a notification is not explicitly directed by the producer to specific consumers, but is simply multicast to consumers whose subscriptions match the notification description.
- The Elvin paradigm is independent of agent toolkit – there are APIs implemented for C, C++, Perl, Python, and Java languages, and so can be used for communication between agents built using different toolkits.

We discuss design, implementation and experiments that were conducted to investigate the potential and practicality of integrating a publish-subscribe communication mechanism such as Elvin with mobile agents, as first suggested in [3].
The rest of this paper is organized as follows. Section 2 discusses relevant background concepts. Section 3 discusses our implementation together with technical hurdles. Using the event-based mechanism, however, has one drawback, which is the loss of messages during agent migration. Section 4 describes the experiments we conducted on event-based communication between agents from two different toolkits, and the extent of message loss due to agent movement. Section 5 concludes with future work.

2 Background and Related Work

2.1 Publish-Subscribe Model

A common model for performing asynchronous communication in component-based environments is the publish-subscribe model [1]. Generally, entities that wish to send messages "publish" them as events, while entities that wish to receive certain types of messages (or events) "subscribe" (or register) to those events. A publisher of a message is not aware of the recipients requesting that message and might not even be aware of their existence. Similarly, components that receive messages may not be aware of other components that may also listen to the same event and can only receive messages they are registered for. Often, an entity may become both a publisher and subscriber, sending and receiving messages within the system.

To accomplish the event-based model, a separate entity is deployed between the producer and consumer of a message, which decouples the connection between those entities and provides the necessary mechanism for distributing a message to registered subscribers.
We observe the stateless nature of the publish subscribe model, which is characterized as fast, efficient and relatively simple implemented mechanism, and distinguish it from a state-full implementation such as message queuing that uses a different style for performing communication. A weakness of the stateless model is the potential loss of messages. In contrast, message queues use the store-and-forward technique, which guarantees the delivery of messages but results in high latency, greater overheads and slow message delivery [6]. Examples of message queuing systems are IBM's MQSeries [12], BEA MessageQ [13] and a range of applications that are built using the Java Message Service (JMS) [7], such as the Oracle Message Broker [8].
Further distinction of event-based systems is the evolution from subject-based systems, in which a message relates to a group of subjects, towards content-based systems, where a subscriber can search for content that appears within a message. A content-based communication mechanism, used in heterogeneous multi-agent systems can become a powerful means of communication between agents that share similar interests (content) without having prior knowledge of each other.
Several content-based event systems, such as Elvin [5], LeSubscribe [9] and Gryphon [6] exist, offering a selection of services and attempts to overcome the undelivered messages problem [6].

2.2 The Elvin Event Notification System

Elvin provides a scalable and dynamic publish-subscribe event notification mechanism. It is built and used in a client-server architecture, in which an Elvin server is responsible for managing client connections and transferring messages between publishers and subscribers. Consumers of notifications register their interest in specific events with the server. Upon receiving a notification (message) from a producer, the Elvin server forwards the notification to the relevant client subscribers by comparing the message content with the list of subscriptions it holds. As the routing of a message is based on its content, it provides the flexibility to operate in a dynamic environment and is independent of the need to configure information relating to the recipients of a notification. The notification itself is encapsulated within an object and contains a list of key-value pairs. A consumer expresses its interest with a subscription element, which is built with a special subscription language containing predicates.

2.3 Agent Toolkits – Aglets and Grasshopper

Two mobile agent toolkits were used during this study: IBM's Aglets (www.trl.ibm.com/aglets/index_e.htm) and IKV++ GmbH Grasshopper (www.grasshopper.de). Although different in other aspects, both toolkits provide an internal communication mechanism used to send messages between agents imple-

mented in the same toolkit. The Grasshopper communication mechanisms are of particular interest as they provide a variety of messaging facilities including message multi-casting. Both toolkits expose a set of Java classes as an API to facilitate mobile agent implementation.
In Aglets, mobile agents live and act within an environment (object) termed context. The context (located in specific host/s) manages the active agents. Grasshopper's execution environment is more complex and consists of several elements, termed regions, agencies and places.

3 Implementation

Prior to this research, Elvin was successfully used in several applications, such as CSCW environments to achieve mutual awareness between different parties. However, application objects that were utilizing the Elvin mechanism in those applications were static, in the sense that they were located at the same host for their entire lifetime.
In contrast, mobile agents may be active and migrate between different hosts. During the lifetime of a mobile agent, it can execute on several agent places in different times and may need to communicate with other mobile agents as well as with static objects.
As Elvin's original focus is to provide an event-notification based communication for traditional distributed applications and architectures, in which objects that are using Elvin remain in the same location, it was initially unfit to be used as part of mobile code. Hence, new functionality had to be added to facilitate such integration. Furthermore, the newly developed capabilities were targeted to be used by any mobile agent toolkit and not be restricted to a specific one.
The new functionality that supports the use of Elvin in mobile code was developed to integrate easily with the regular mobile agent's code, requiring only minimal changes in the agent's program. As mobile agent toolkits mostly operate on a Java Virtual Machine, development of new functionality was done in the Java Environment.

3.1 Basic Elvin Client Constructs and Mobility Hurdles

Two fundamental constructs for performing Elvin communication are the *consumer* and *producer* objects. These objects are part of the Elvin Client library, which enables *consumer* objects to register callback methods that are activated for specific notifications, and allow *producer* objects to publish new notifications intended for listeners (that use the *consumer* object). The Elvin Server behavior dictates that an application that consumes notifications (by using the *consumer* object) must exist in the same location in which it originally subscribed for the particular notification/s at the time the notification is sent by the server. An Elvin *connection* object describes the communication details and exists for each *consumer* object. Upon receiving an event, the server will broadcast a notification for that event to subscribing objects, in the location

in which the original subscription took place (and as maintained by the Elvin *connection* object).
Mobile agents cannot use these Elvin constructs since they often change their location and still require receiving notifications. In such a scenario, the Elvin server is not aware of their new location and continues to send messages to the old location. It is also infeasible for them to return to their original subscribing location, and even if they do so, they will lose many of the notifications sent to them, while in transit or executing in other nodes. We will discuss solutions to this problem later.
In Figure 1, which illustrates this problem, a mobile agent first subscribes for events (stage 1), and then migrates to another location (stage 2). An Elvin Server contains a subscription registration from the agent, and a connection that is identified with the old location, thus sending new notifications to the wrong host.

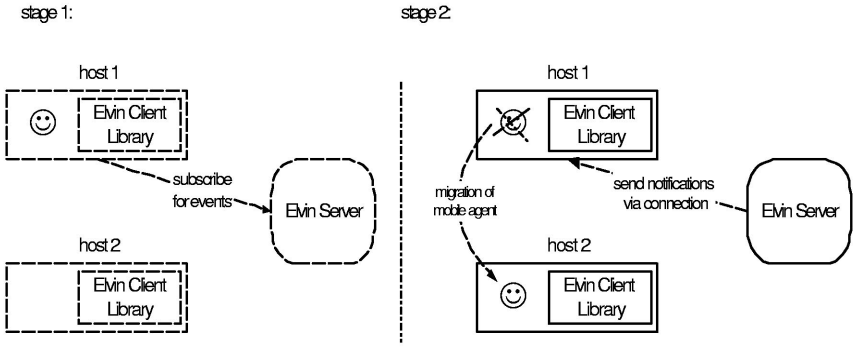


Fig. 1. The connection problem between the Elvin Server and a mobile agent.

A second important mobility problem is concerned with the nature of migration of mobile agents. While in transit, a mobile agent has no concrete existence on a particular host and is therefore unable to receive Elvin messages. We consider this problem in more detail later.
A third hurdle in implementing a solution for mobile agents is the fact that all Elvin Client objects are implemented as non-serializable objects. As such, they cannot migrate as part of the mobile agent code to new locations. This imposes obvious restrictions on the way mobile agents can use existing Elvin objects.

3.2 MobileConsumer

To solve the mentioned mobility problems, new functionality, represented by a *MobileConsumer* class was developed and added to the Elvin Client Library. The *MobileConsumer* class supports Java Serialization, and thus, can be transported to different hosts as part of the mobile agent. The functionality implemented in this class follows the approach suggested in [3], in which an agent removes all existing subscriptions before migrating to a new location and re-subscribes to the same notifications after arriving at the new location.

The class embeds this kind of behavior and performs the required operations on behalf of its client (the mobile agent). A *MobileConsumer* object keeps information such as a serializable representation of Elvin subscriptions and Connections, and is implemented as a single object per client. Consequently, a mobile agent client needs only to use the *MobileConsumer* object in two predefined states: before the migration, and right after the migration.
Two public methods are provided for this purpose:

```
NotifyPreMigration()
NotifyNewLocation(…)
```

The client uses the first function just before migrating to a new location, and the second is activated immediately after arriving at the new location. Ideally, these functions are implemented in callback functions that are activated just before and after migration (by the agent's toolkit execution environment). Such a feature is common in most mobile agent toolkits. The following code snippet demonstrates the use of *MobileConsumer* and other Elvin classes in a basic scenario, in which a mobile agent subscribes for notifications:

```
MobileConsumer mc =
  new MobileConsumer(ElsvinServerURL);

public void onCreation(Object itin) / init(Object[] crea-
tionArgs){
     Subscription sub = new Subscription(strSubscription);
    // NotificationHandler → class to handle
    // notification events
    NotificationHandler nh = new NotificationHandler();
    sub.addNotificationListener(nh);
    mc.addSubscription(sub);
}

public void onDispatching(MobilityEvent m) / beforeMove(){
    mc.notifyPreMigration();
}

public void onArrival(MobilityEvent e) / afterMove(){
    NotificationHandler nh = new NotificationHandler();
    mc.notifyNewLocation(nh, strSubscription);
}
```

The '/' separates the different names in the Aglet and Grasshopper toolkits for callback methods of similar purpose.

4 Experimental Evaluation

4.1 Communication between Mobile Agents of Different Toolkits via Elvin

The current limitation on various mobile agents communication mechanisms is their inability to perform outside the agent's toolkit environment. As previously described,

implementing the *MobileConsumer* class in the Elvin Client Library enables easy integration of Elvin Client on different mobile agent platforms. Consequently, different vendors' mobile agents can communicate (peer to peer), share knowledge and receive broadcasted notificaions, all with the use of Elvin.

The following experiment examined the capability to perform communication via Elvin by migrating mobile agents. The communication was performed between agents running on two different platforms: Grasshopper and Aglets. The architecture consisted of four Grasshopper agencies and four Aglet servers representing eight different hosts. Two mobile agents, one for each toolkit, were migrating between these hosts (Grasshopper agents to Grasshopper agencies and Aglets to Aglet servers). The mobile agents were listening and sending Elvin notifications to each other, and to a static service called *Simulation Monitor*, which was also receiving Elvin notifications.

The *Simulation Monitor*'s task consisted of obtaining the state of the experiment at a given time, in terms of the agents' locations and messages sent and received by those agents. This information could then be visually expressed and verified against the expected results.

An Elvin Server process was also running to facilitate the communication. The Elvin Server conducted all communications between the agents themselves and the monitor service. Elvin does not restrict the amount of subscriptions applications can have, or the amount of events they produce.

As a result of the messages exchanged via Elvin, the *Simulation Monitor* is always aware of the current locations of the agents, as well as to the messages received by them (the monitor can also obtain the messages sent between the agents by simply subscribing to them). However, in this experiment, the monitor serves as a control unit to provide us information on the agents' actual capabilities of communicating and receiving messages.

Figure 2 (A, B and C) illustrates the basic communication scenarios that together formed the whole process in the experiment. In Figure 2A, at time t1, the agents and *Monitor* service subscribe for events. Then at time t2, the agents activate the `NotifyPreMigration()` functionality. On arrival to a new location (Figure 2B), agents activate the `NotifyNewLocation()` method call (at time t3). At time t4, the agents send Elvin notifications - publishing their arrival (for the monitor) and sending Hello messages to each other.

In Figure 2C, agents and the monitor accept Elvin notifications (time t5). Upon receiving a peer notification, an agent sends an Elvin message to the monitor (time t6).

Successful results of the experiment have proven the ability of Elvin to serve as a communication mechanism between mobile agents that perform migration, independent of platform considerations.

4.2 Measuring Message Delivery

Most Elvin related procedures are activated just before or after migration. We conducted performance measurements of the total time of Elvin related procedures vs.

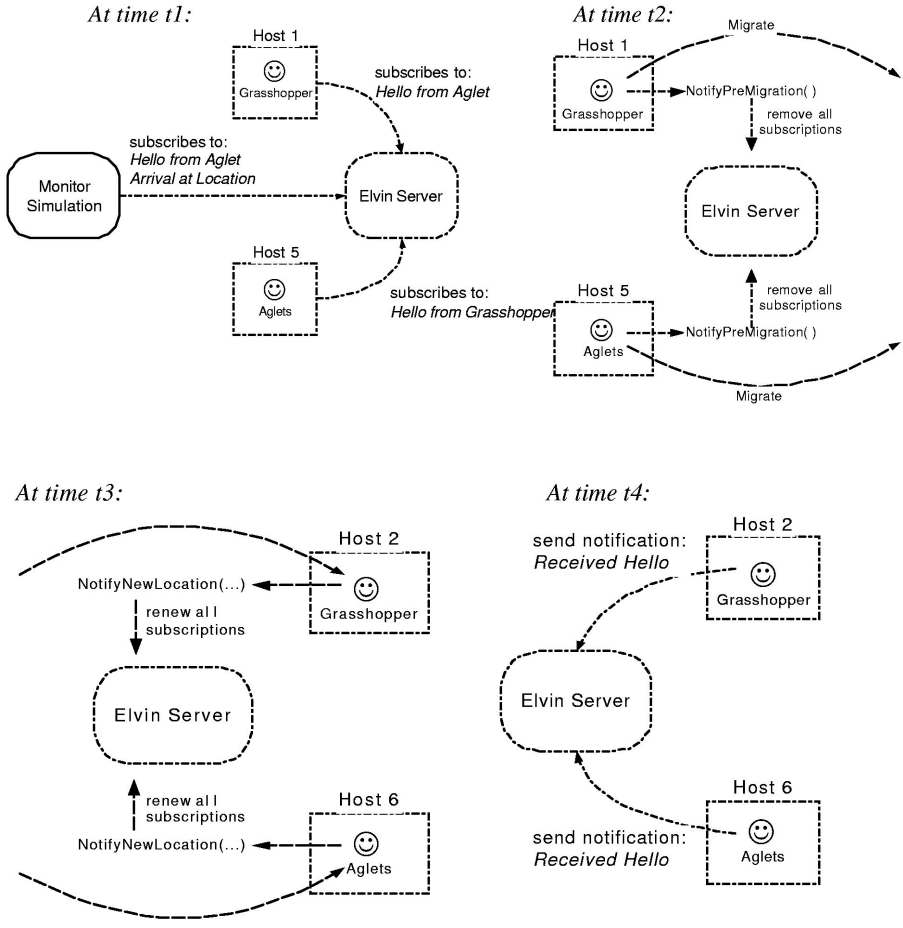


Fig. 2. Figure 2 (A and B). At time t1, t2, t3 and t4. (t1<t2<t3<t4<t5<t6<t7)

the total time of migration (excluding Elvin procedures) in Grasshopper mobile agents in a single machine. The results of the test suggest that a relatively small amount of time is consumed by Elvin procedures compared with migration time. The average ratio of the time for Elvin procedures to migration time was about 1 to 16. Despite the seemingly good performance of Elvin, the results imply a relatively long duration of migration, which can result in lost messages. Mobile agents are not responsive during migration (transfer of code and state); hence messages that are sent to those agents during that time are lost. A relatively long migration time can mean that considerable amount of messages may be lost during the life-cycle of an agent. Other factors that also affect the delivery of messages are bandwidth and communication latency, and duration of stay of an agent in a location. The greater the bandwidth the less the migration time (i.e. the time an agent spends in migration), and consequently, the fewer

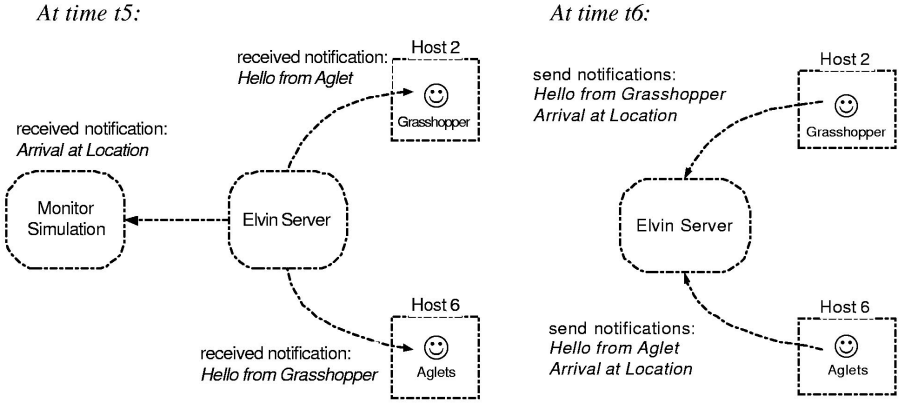


Fig. 2C. At time t5, t6

the number of lost messages. Similarly, a longer stay in a location may also reduce the average number of lost messages.

This hypothesis can be described in the following way. The average percentage of lost messages is expected to be influenced by migration time, and the average delay of agents at a location (i.e. the length of time the agent stays at a location on average). An increase in the migration time is expected to increase the amount of lost messages, as the agent then spends more time in a non-responsive (transit) state. Conversely, the smaller the migration time, the greater the chance that a message will find the agent in a responsive state. Migration time itself is expected to be influenced by bandwidth. A low bandwidth will cause an increase in migration time, as agent migration is the transfer of a stream of code and state between two hosts.

Agents' delay at a location is expected to have a negative effect on the amount of lost messages. The greater the delay in the location, the greater the chance that a message will find the agent in a responsive state. Consequently, on average, fewer lost messages are expected when agents stay longer at a location.

An experiment that examines the effect of various parameters on the amount of lost messages was conducted by simulating mobile agent behaviour.

The experiment consisted of six different hosts located on four physically separated machines, five Aglets mobile agents, an agent *creator* process, and a *monitor* service process. Mobile agents, created by the *creator* process, migrate to other hosts, and send Elvin notifications to each other. In order to produce a close-to-reality simulation, the agents' behaviors are randomized. Each agent obtains a randomly chosen migration path between the hosts and sends arbitrary number of messages to randomly chosen agents in each new host it arrives at. Consequently, each agent operates a different migration path and receives a different total number of messages from different agents. Agents receive messages from other agents by subscribing to Elvin. An agent can utilize content-based addressing, and subscribe to messages with certain attributes (without explicitly specifying the identity of agents it would like to receive messages from), or might use agent identifiers in the subscription to subscribe to messages (in-

tended for itself) from a particular agent (thus emulating point-point message delivery).

The *Monitor* service subscribes to all the messages produced by each agent. Hence, it continuously receives messages that are sent between the agents. The *Monitor* aggregates the total messages sent to each agent, and compares those with the number of messages that were actually received by the agents. A mobile agent internally counts messages it receives and at the end of its migration process sends the count to the *Monitor*, via Elvin.

Two parameters - delay per location, i.e. the time spent by an agent at a location, and (bandwidth) delay in agent migration, i.e. the time spent by the agent in transit – were dynamically configured to measure their effect on the percentage of lost messages.

Each agent would calculate a random location delay time based on an upper bound, for every new location it arrives at. By choosing a higher upper bound for an experimental run, we would expect a greater average delay per location for the agents.

Delay in agent migration was simulated by postponing the notification of an agent's new location to Elvin, thus appearing to the system as a delay in migration.

Communication between agents was performed in a networked environment using a remote Elvin server, denoting a true representation of communication bandwidth.

4.3 Results and Interpretation

An initial set of experiments was conducted, measuring different delays per location while simulating high bandwidth (small delay in migration). The results indicate a clear trend of decrease in lost messages for greater delays (of agents) per location.

Figure 3 shows the effect of the agent's location delay (i.e. the upper bound) on the average percentage of lost messages. The greater the average delay, the fewer messages lost.

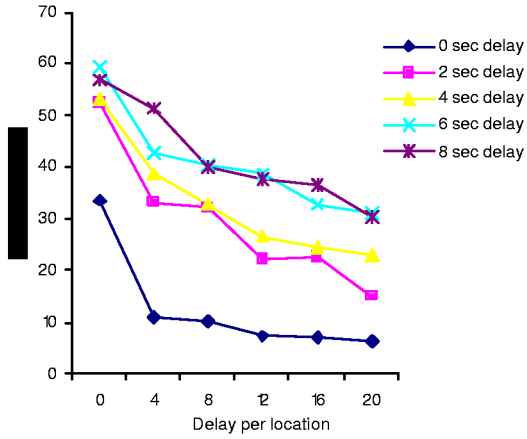


Fig. 3. Effect of different migration delays

A second stage of the experiment added a simulation of low bandwidth to examine the influence of long migration periods on the percentage of lost messages. This operation did not affect, however, the speed of Elvin communication between the agents (since the messages are small). Bandwidth's effect was than examined with added delays in migration time ranging between 0 and 8 seconds. The figure also shows the effect of different migration delays (i.e., the different graphs), against delay per location.
On average, the rate of decrease of lost messages is initially high and gradually declines, often resembling an asymptotic curve. The increase in migration delay leads to an increase in the percentage of lost messages.

5 Conclusions and Future Work

We have proposed a content-based publish subscribe communication genre for mobile agents in heterogeneous multi-agent systems, which is useful in the case of exchanging and disseminating short communications about events. We have demonstrated the applicability of the idea by experimenting with the Elvin system and developing functionality that enables the integration of Elvin with mobile code. Our current implementation suffices when guaranteed point-to-point delivery of messages is not required, e.g. applications where information about events are disseminated and whoever is listening might take advantage of it. On one hand, this is not a problem as point-to-point delivery between specific senders and receivers is not the intention of this genre, and there are other communication genres (e.g., the traditional agent communication language implementations) that handle that. Other event-based systems such as Gryphon [6] have tried to address the message loss issue, with a limited amount of success. It can also be argued that guaranteeing the delivery of messages is not part of the stateless publish subscribe paradigm and should be dealt with in the application layer. We have considered three solutions in [14] for the problem of messages not being received by the agent while in transit.
In conclusion, we mention the DEMS (Distributed Event Messaging System) [4] developed for mobile agent monitoring, control, and communication. DEMS is based on the Java event model of event objects, event listeners, and event sources, and is used with the EMAA toolkit. Our Elvin-based approach can be used for the same purpose – to monitor and control mobile agents – but we leverage on the Elvin publish-subscribe infrastructure, including Elvin's scalable architecture of being able to handle thousands of events efficiently, and the rich subscription language. In addition, our approach is toolkit independent, and in principle, can be programming language independent, as long as there is an Elvin client library for that language.

References

[1] Dingel, J., Garlan, D., Jha, S., and Notkin, D. Towards a Formal Treatment of Implicit Invocation. *Proceedings of the 1997 Formal Methods Europe Conference*, 1997. Available at http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/implicit-invoc-fme97.html

[2] Loke, S.W. An Overview of Mobile Agent Technology for Distributed Applications: Possibilities for Future Enterprise Systems. *Informatica: An International Journal of Computing and Informatics*, Vol. 25(2), July 2001, pp 247–260.

[3] Loke. S.W., Rakotonirainy, A., and Zaslavsky, A. Enabling Awareness in Dynamic Mobile Agent Environments (short paper). *Proceedings of the 15th Symposium on Applied Computing (SAC 2000)*, Como, Italy, March 2000, ACM Press.

[4] McCormick, J., Chacón, D., McGrath, S. and Stoneking, C. A Distributed Event Messaging System for Mobile Agent Communication, March 2000, TR-01-02. Available at http://www.atl.external.lmco.com/overview/papers/986.pdf

[5] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. Content Based Routing with Elvin4*,Proceedings AUUG2K*, Canberra, Australia, June 2000
Available at http://elvin.dstc.edu.au/doc/papers/auug2k/auug2k.pdf

[6] Sumeer Bhola Robert Strom Saurabh Bagchi Yuanyuan Zhao, Joshua Auerbach fsbhola, robstrom, sbagchi, yuanyuan, Exactly-once Delivery in a Content-based Publish-Subscribe System, IBM T.J. Watson Research Center

[7] Sun Microsystems. *Java Message Servic*e, November 1999.

[8] Oracle Message Broker 1.0, Datasheet October 1999

[9] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, Matching Events in a Content-Based Subscription Systems, *Proc.18th ACM Symp. Principles of Distributed Computing (PODC '99), 1999.*

[10] F. Fabret et al., Filtering Algorithms and Implementation for Very Fast Publish /Subscribe Systems*, SIGMOD Conf., 2001.*

[11] Labrou, Y.; Finin, T.; and Peng, Y. 1999. The interoperability problem: Bringing together mobile agents and agent communication languages. In Ralph Sprague, J., ed., *Proceedings of the 32nd Hawaii International Conference on System Sciences*. Maui, Hawaii: IEEE Computer Society.

[12] IBM MQSeries online overview and documentation
http://www-3.ibm.com/software/integration/mqfamily/

[13] BEA MessageQ online overview and documentation
http://edocs.bea.com/tuxedo/msgq/

[14] Padovitz A., Zaslavsky A, Loke S.W. Awareness and Agility for Autonomic Distributed Systems: Platform-Independent Publish-Subscribe Event-Based Communication with Mobile Agents. Accepted for publication in the *1st International Workshop on Autonomic Computing Systems*, to be held at DEXA 2003.