Context-Based Addressing: The Concept and an Implementation for Large-Scale Mobile Agent Systems Using Publish-Subscribe Event Notification

Seng Wai Loke, Amir Padovitz, and Arkady Zaslavsky

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

Abstract. We introduce the notion of context-based addressing, i.e. the ability to refer to and send messages to a collection of agents based on their current context, without knowing the precise identities of the agents. We describe a simple implementation of context-based addressing for mobile agents using Elvin, a publish-subscribe event notification system, as a proof-of-concept, and to investigate the feasibility of the event-based paradigm for implementing context-based addressing for mobile agents.

1 Introduction

Human-beings can be identified and referred to in numerous ways, ranging from our names, our nick-names, our titles, our roles, to where we live. The ability to address or refer to software agents using a variety of methods can provide flexibility and abstraction for system developers, particularly when the system being built is dynamic and large, involving a huge number of agents.

A dimension of dynamic behaviour is the property of agents being able to move from one host to another to perform computations. While not all agents require such a property, agent mobility has been recognized as beneficial in a number of large scale distributed applications, including information-centric applications [5] and a sizable number of mobile agent toolkits[1] have been developed for mobile multiagent applications. Proposals for the use of mobile agents in network management are also numerous (e.g., [1]). More recent applications of mobile agents include large networks of embedded systems such as wireless ad hoc sensor networks [4].

Large systems of mobile agents over large distributed environments are generally difficult to program and various techniques have been introduced. In this paper, we consider one aspect of programming mobile agents, i.e. the ability to refer to and send

[1] See http://mole.informatik.uni-stuttgart.de/mal/preview/preview.html

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 274–284, 2003.

messages to a collection of agents based on their current context, without knowing the precise identities of the agents. By agent's context, we refer to the situation of an agent as is made known by the agent. For example, suppose we have a large number of mobile agents roaming over hosts distributed throughout an intelligent building performing various functions (e.g., as in the Hive project http://hive.sourceforge.net/hive-asama.html), and, without explicitly addressing each agent, we want to send a message to all agents which are currently on hosts situated on the third floor, or we want to send a message to all agents launched by (and belonging to) John Smith, or we want to send a message to all agent's currently running on hosts with less than one gigabyte of available storage, or we want to send a message to all agents currently on hosts which provide a directory service, or we want to send a message to all agents which have not yet completed their tasks and are currently running on hosts with connections whose network traffic is increasing, or we want to send a message to all agents currently running on a host which will be disconnected from other hosts in the next three minutes. All these messages target agents according to the context which the agents are in, and the way we refer to these agents is via the current context rather than their names (though it is possible to find their names with such a context-based query as "send me your name if you are on a host on the third floor"). We think that such *context-based messaging* will be useful for distributed monitoring, management and control applications.

We propose a mechanism for supporting such context-based addressing, and demonstrate our ideas assuming that agents have access to context-information that will be used in addressing the agents, and that the agents proactively report their current context to a central server. What exactly the context information will be depends on the application semantics.

Our approach leverages on a publish-subscribe event-based communication system called Elvin. The publish-subscribe event paradigm has been useful for asynchronous communication between loosely coupled components of distributed systems. 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 a generic representative of the class of event notification systems.

Elvin provides implicit invocation [2]: 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. As we describe later, we can use content-based addressing as a means of selecting agents to send messages to.

The rest of this paper is organized as follows. Section 2 discusses relevant background concepts. Section 3 discusses our initial implementation, integrating Elvin's publish-subscribe communication mechanism with mobile agents, as first suggested in [6]. Section 4 discusses the notion of context-based addressing for collections and subcollections of agents that is based on content-based addressing. Section 5 concludes with future work.

2 Background

2.1 Publish-Subscribe Model

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.

2.2. The Elvin Event Notification System

Elvin 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. Client Applications use the Elvin's client code and API in order to produce and consume information in a publish-subscribe fashion [8].

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 and not on intended recipients, 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. The notification element supports several data types, such as integers, floating points, strings etc. A consumer expresses its interest with a subscription element, which is built with a special subscription language containing simple logical expressions [3].

Consider the following subscription expression taken from [3]:

```
(TICKERTAPE == "elvin" || TICKERTAPE == "Chat") && ! re-
gex(USER, "[Ss]egall")
```

This subscription expression will match any notification whose `TICKERTAPE` field has the string value "elvin" or "Chat" except those whose `USER` field also matches the regular expression "[Ss]egall". This subscription would match the following notification example:

```
TICKERTAPE: "Chat"
USER:       "alice"
TICKERTEXT: "hello sailor"
TIMEOUT:    10
Message-Id: "07cf0b15003409-5i3N7XDKbPVaQ-28cf-22"
```

3 Integrating Mobile Agents with Elvin

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).

Another important Elvin Client construct is the *subscription* object, which describes the notification, and is sent to the Elvin Server.

The Elvin Client Library is used by applications to produce and receive notifications from other applications, whereas the Elvin Server process is controlling, coordinating and managing all communication between the clients. 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 much 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.

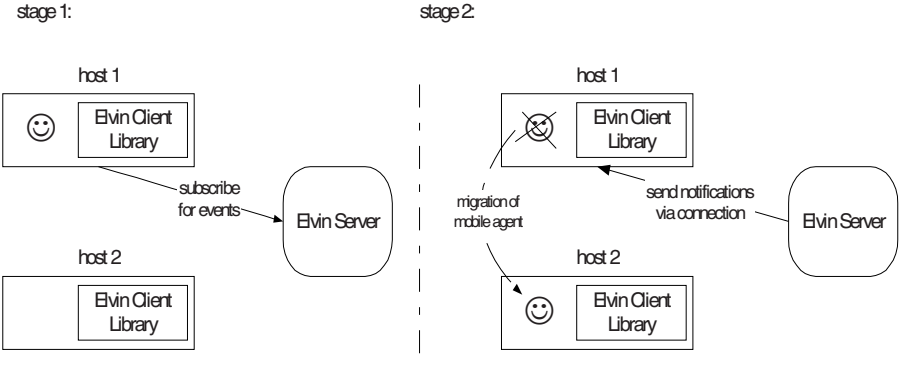

Fig. 1. The connection problem between the Elvin Server and a mobile agent

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-serialisable 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 [6], 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[] creationArgs)
{
     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.

Using the event-based mechanism of Elvin is lightweight and efficient even for large volumes of messages, but has one drawback, which is the loss of messages during agent migration. We conducted experiments on the effect of agent transit on message lost in [7], and saw that the percentage of messages lost or not received by an agent while in transit is on average relatively low. In one sense, this is not an issue as we are not intending the event-based mechanism primarily as a guaranteed point-to-point communication device (we already have other means of doing so) but for large-scale event dissemination. Whether a small percentage lost of messages is significant depends on the semantics of the application, but results in the rest of this paper will work for other event systems comparable with Elvin. We also note that if message lost cannot be tolerated in an application, the problem can be easily solved by using a stateful, but more heavyweight event notification system.

4 Context-Based Addressing for Collections and Subcollections of Agents

Given that mobile agents can now subscribe to Elvin and receive messages (i.e. notifications) matching the subscriptions, we can employ this mechanism for context-based addressing of agents in the following way. Each agent subscribes to Elvin for messages that relate to its context. An agent does this by using particular keywords in its subscription expression that relates to its context, effectively reporting its context to Elvin (i.e., the Elvin server). For example, an agent uses the following expression to "listen for" messages intended for agents who are low in credits and are currently on hosts which have less than 5GB of free disk space:

```
(CREDITS == "low" && DISKSPACE == "< 5GB")
```

Such a subscription expression will match the following notification, which in this example carries a command to agents in such a situation to come home:

```
CREDITS: "low"
DISKSPACE: "< 5GB"
COMMAND: "come_home"
Message-Id: "07cf0b15003409-5i3N7XDKbPVaQ-28cf-22"
```

Note that the attribute names, the possible values for each attribute, and their semantics must have been pre-defined, for example, based on an ontology of context attributes (perhaps pre-defined for given applications). We also note that the sending of the command notifications need not know the identifiers of those agents in the situation described above – such is the power and advantage of implicit invocation (or content-based addressing). Moreover, a "name" for the collection of agents in the above described situation has been effectively created. Such a "name" can be used to send a message to these agents, asking them to report their names, i.e. we can query to find out the which agents are currently in a particular context.

One can also refer to subcollections in that collection of agents by adding additional constraints and leaving it to the agent to check if the message received applies to itself. For example, suppose the notification was the following:

```
CREDITS: "low"
DISKSPACE: "< 5GB"
GROUP: "Tom"
COMMAND: "come_home"
```

Message-Id: "07cf0b15003409-5i3N7XDKbPVaQ-28cf-23"

The agent will still receive such a message as it matches with the above subscription, but in checking that it does not belong to Tom's group, it might decide to ignore such a message. Alternatively, the agent can shift such filtering to the Elvin server (reducing some network traffic) by using a more elaborate subscription expression that specifies it belongs to Jane's group (and so Elvin will not forward to the agent messages intended for Tom's group), such as the following:

```
(CREDITS == "low" && DISKSPACE == "< 5GB" && GROUP == "Jane")
```

Hence, an agent can inform Elvin of its current context and in this way receive messages suited to its context. An agent can effectively modify its subscription (by unsubscribing and resubscribing) updating the context information it provides to Elvin. Subscriptions carry information about agents' current context, and notifications, via particular attributes and values, can target agents in specified context – it is in this way that the content-based addressing of Elvin is, when context attributes are utilized (and when agents truthfully report their context to Elvin), viewed as context-based addressing. Other examples of context attributes that might be used include the location of the agent, the status of the agent's task execution, a fragment of the agent's state, a description of the agent's current execution environment (e.g., services available), and bandwidth for relevant connections (assuming discrete values for these context attributes).

4.1 Ad Hoc Namespaces

The above suggests that it is possible to define namespaces ad hoc over collections of agents. For example, one can create a namespace that relates to the geographical location of hosts. For example, suppose we consider the division of a building into logical areas such as the following. A building (call this Building1) has five floors, i.e. Floor1, Floor2, Floor3, Floor4 and Floor5. In each floor, we have five rooms numbered Room1, Room2, etc up to Room5. Suppose that there are one or more computers in each room, and that agents residing on a computer can find out the logical area (i.e., there is a means to map IP addresses to logical areas) they are currently in, and send a subscription to Elvin, with an attribute identifying the logical area they are currently in. We can then send a message to all agents in Room3 via notifications such as

```
LOGICAL-AREA: "Building1/Floor3/Room3"
COMMAND: "come_home"
 …
```

Agents in Floor3 can subscribe to all messages for Floor3 agents via a subscription of the form:

```
(begins-with(LOGICAL-AREA, "Building1/Floor3/") || …)
```

As agents move and update their context information with Elvin via subscriptions, the collection of agents referred to with the phrase "Agents in Floor3" might change.

Several namespaces can be used jointly. For example, apart from partitioning agents according to their geographical location, we might partition agents according to who their owners are. We can then refer to "Agents currently in Floor3 AND who belong to Seng", and send messages to such agents. With large numbers of agents in large numbers of hosts and ubiquitous embedded devices, such "names" provide a convenient means to refer to whole collections of agents. Such "names" as "Agents currently in Floor3 AND who belong to Seng" can be used within programs in applications involving agents or by users who want to control the agents directly (for example, a user's command such as "Agents currently in Floor3, move to John's machine" can be issued).

Our approach of context-based addressing relies on the agents proactively reporting to Elvin their current context via a subscription. However, this is not mere reporting as a subsciption carries with it the semantics of inviting messages (or notifications) that are relevant to the context specified, i.e. the agent via such a subscription is effectively saying to Elvin "This is my current context. Please forward messages (if any) that relate to this context" and any subsequent messages matching the subscription will be forwarded. Note that the agents are sending subscriptions to report their contexts rather than sending notifications. If the agents send notifications, than we have a monitoring model where agents notify subscribers about their contexts. However, we are implementing a messaging model, where agents are receiving messages according to their contexts, and hence, agents send subscriptions and not notifications. While a mix of such models can be employed in an application, our focus has been on a messaging model. We leave it up to the designer of the agents to ensure such behaviour in the agents he/she creates. The required efficiency of agents, network capacity, and application semantics will determine how often the agent should (re)issue such subscriptions[2].

4.2 A Programming Construct

There are two ways such context-based messaging can be used. One way is when the user needs to send commands to a collection of agent, referring to them by context. In such a case, a user interface can be built which will allow a user to specify such commands. The other way is when a software agent needs to send a context-based message to other agents. We describe here a programming construct for this purpose, as follows:

```
agents_in_context(<context expression>)#<message>
```

[2] Elvin is made to handle large numbers of such small messages efficiently.

whose semantics is to send `<message>` (e.g. a string) to the agents referred to by the `<context expression>`, which is a set of attribute-values as shown in the previous subsection. Using Elvin, executing this construct generates an Elvin notification to be forwarded to the Elvin server. Given a context expression, the set of agents referred to depends on the time the construct is evaluated since agents' contexts are expected to vary over time.

Such a construct can be implemented as a Java class for use in building applications.

5 Conclusions and Future Work

We have introduced the notion of addressing and messaging agents by the context or situation which they are currently in, which we called *context-based addressing*. We have also illustrated this concept via a simple implementation using Elvin. There are drawbacks in using Elvin, it being a stateless system so that messages might be lost when agents are in transit. However, this has not been the focus of this paper and other stateful event systems can be used. What we have shown is that the publish-subscribe event model can be employed to implement context-based addressing, provided the agents are built to proactively report their context to the event system via subscriptions (in effect expressing their desire to receive messages intended for their present context). In terms of context management, this means that the agents have to be programmed to know what context to report, and how often to report. A principled methodology is required in which the internal architecture of the agent is structured into a context-reporting module and application-specific modules. An alternative is to have other components report about the agents, but these components need to be build in some principled manner, and it would mean agents lose autonomy in deciding what and when to report. Applications of such context-based addressing have been suggested in the introduction, and our future work involves utilising such addressing in smart environment systems. Further experimentation will be required to test the scalability of our approach. The ability to network Elvin servers provide a means to support highly distributed scenarios. New versions of Elvin supports security in that only certain consumers (with the right key) can decipher the messages of certain producers.

References

1. Bieszczad, B. Pagurek, and T. White, "*Mobile Agents for Network Management*", IEEE Communications Surveys, Vol. 1, No. 1, Fourth Quarter 1998.
2. J. Dingel, D. Garlan, S. Jha, and D. Notkin. 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
3. DSTC. The Elvin Subscription Language. http://elvin.dstc.edu.au/doc/esl4.html

4. H. Qi, F. Wang, "Optimal itinerary analysis for mobile agents in ad hoc wireless sensor networks," *The 13th International Conference on Wireless Communications*, vol. 1, pp.147-153. Calgary, Canada, July, 2001.
5. M. Klusch, F. Zambonelli (ed): Cooperative Information Agents – Best Papers pf CIA 2001. Intl. Journal of Cooperative Information Systems. 2002 Vol. 11, No. 3 and 4.
6. 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.
7. Padovitz, A., Loke, S.W., and Zaslavsky, A. Using Publish-Subscribe Event Based Systems for Mobile Agents Communication. Submitted.
8. Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. Content Based Routing with Elvin4*, Proceedings AUUG2K*, Canberra, Australia, June 2000 URL: http://elvin.dstc.edu.au/doc/papers/auug2k/auug2k.pdf