Communicative Acts of Elvin-Enhanced Mobile Agents

Seng Wai Loke and Arkady Zaslavsky
School of Computer Science and Software Engineering
Monash University, Caulfield East, VIC 3145, Australia
{ swloke@csse.monash.edu.au, a.zaslavsky@monash.edu.au}

Abstract

Elvin is a scalable and efficient publish-subscribe event notification system. We provide an interpretation, in terms of communicative acts, of the Elvin messages that are used by mobile agents in order to utilize Elvin services. We provide a basis for integrating Elvin services within a multi-agent community, i.e. agents can interact with Elvin at the agent communication language level, and show how to implement group communicative acts using Elvin. We believe that the proposed approach enhances agent communication mechanisms with the general-purpose publish-subscribe paradigm.

1. Introduction

We provide an interpretation, in terms of *performatives* or communicative acts [4], for the messages used by the mobile agents to utilize Elvin services. While implementations of publish-subscribe models like Elvin 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. We provide a means for mobile agents to use Elvin services at the high abstraction level of agent communication or the knowledge level, even for group communication.

2. The Elvin Event Notification System

Elvin provides a scalable and dynamic content-based 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. Client Applications use the Elvin's client code and API in order to produce and consume information in a publish-subscribe fashion [5].

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 [1].

Consider the following subscription expression taken from [1]:

```
(TICKERTAPE == "elvin" || TICKERTAPE == "Chat")
&& ! regex(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. ACL Messages for Interaction with Elvin

We now consider how the performatives in [2] can be used to interpret Elvin messages.

3.1. Subscriptions to Elvin

Three performatives that have semantics similar to that of subscribing to receive event notifications are as follows [2]: **`request-when`**, **`request-whenever`**, and **`subscribe`**.

In Elvin, subscriptions (with associated subscribers) once with the Elvin server are matched against notifications received, and notifications are sent to the subscriber as long as the subscriber does not unsubscribe a subscription. Hence, over time several notifications

Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology (IAT'03)
0-7695-1931-8/03 $ 17.00 © 2003 IEEE

might be sent in response to a subscription. Hence, `request-whenever` and `subscribe` are semantically closer than `request-when` in this respect to an Elvin subscription.

Moreover, matching of notifications to subscriptions is based on attribute-values and the subscription expression, as described earlier, and hence, Elvin itself does not make assumptions about the nature of the event being subscribed to – whether the event concerns a change in a particular object, or whether the event concerns some proposition having become true. `subscribe` requires that its content be a descriptor (a reference) to some object whose change of which is being subscribed to be notified, whereas `request-whenever` requires that its content be a tuple describing an action description and a proposition. Thus, an Elvin subscription maps naturally to a `request-whenever` message where the action is to send a notification to the subscriber, and the proposition is the following: the given Elvin subscription expression matches an Elvin notification, where the subscription expression is provided in the content of the `request-whenever` message.

For example, suppose an agent `i` sends a subscription expression `e` to the Elvin server (wrapped up as an agent named `es`). This subscribe action is expressed by sending (schematically) the following message:

```
(request-whenever
 :sender (agent-identifier :name i)
 :receiver (agent-identifer :name es))
 :content
  "((action (agent-identifier :name es)
      (inform
        :sender (agent-identifier :name es)
        :receiver (agent-identifier :name i)
        :content \"(notification n)\"
      ))
    (matches (subexp e) (notification n))
  )
  "
)
```

The content of the `request-whenever` message is a pair (<action> <proposition>), where <action> is to send an `inform` message, which encapsulates the notification `n`, to the subscribing agent `i`, and the <proposition> is `(matches (subexp e) (notification n))`, i.e., that the notification `n` matches the subscription expression `e`. The Elvin server internally implements such a matching algorithm, and so, this predicate is simply a model of what the Elvin server already does. The semantics of the performative `request-whenever` is such that the `inform` message will be sent whenever the proposition is true.

We note the following from the FIPA specification of `request-whenever` [2]:

"No specific commitment is implied by the specification as to how frequently the proposition is re-evaluated, nor what the lag will be between the proposition becoming true and the action being enacted."

In the case of Elvin, a notification is forwarded (as fast as Elvin can do it) to the subscriber whenever there is a match with a received notification. Hence, if only X notifications have been received by the Elvin server, and Y (<= X) of these match a subscription, Y `inform` messages will sent out from the Elvin server to the subscriber, one for each notification.

An agent can unsubscribe from Elvin by sending a `cancel` ACL message, whose content contains an action expression denoting the action that is no longer needed, which in this case is the `request-whenever` (speech) action. For example, agent `i` sends the following message to `es` to cancel the subscription due to the previous `request-whenever` message.

```
(cancel
  :sender (agent-identifier :name i)
  :receiver (agent-identifer :name es))
  :content
    "(action (agent-identifier :name i)
       (request-whenever
         :sender (agent-identifier :name i)
         :receiver (agent-identifier :name es)
         :content
          \"((action (agent-identifier :name es)
               (inform
                 :sender
                   (agent-identifier :name es)
                 :receiver
                   (agent-identifier :name i)
                 :content \"(notification n)\"
               )
              )
              (matches (subexp e)
                       (notification n)
              )
            )\"
       )
     )"
)
```

The idea is that agent `i` cancels its own previous `request-whenever` action.

3.2. Notifications to Elvin

As for notifications from the Elvin server to subscribers, we have seen that `inform` messages can be used. For notifications from producers to the Elvin server, the choice of performatives should highlight the fact that the Elvin server is only an *intermediary* between producers and consumers. In this regard, two candidate performatives from the FIPA Communicative Act Library [2] for notifications to the Elvin server are: **`propagate`** and **`proxy`**.

proxy stands out from propagate in that notifications to Elvin are not intended for Elvin but for subscribers. The content of a proxy message should be as follows: A tuple of (1) a descriptor, that is, a referential expression, that denotes the target agents, (2) an ACL communicative act, that is, an ACL message, to be performed to the agents, and (3) a constraint condition for performing the embedded communicative act, for example, the maximum number of agents to be forwarded, etc. For example, an agent i sends a notification n (as the proxy message shown below) to the Elvin server (named es), where the contents of the message contains (1) a descriptor defining the target agents that agent i would like the Elvin server to forward the notification to, which is the set of agents who subscribed with some subscription expression (denoted by the variable e, written ?e) which matches the notification n, and (2) the action that the Elvin server is to do, which is to inform the appropriate subscribers of the notification. No constraints are specified.

```
(proxy
  :sender (agent-identifier :name i)
  :receiver (agent-identifier :name es)
  :content
    "( (all ?x (and
                 (subscribed
                    (agent-identifier :name ?x)
                    (subexp ?e)
                 )
                 (matches (subexp ?e)
                          (notification n)
                 )
               )
       (action (agent-identifier :name es)
         (inform
           :sender (agent-identifier :name es)
           :content \"(notification n)\"
         )
         true
       )
    )"
)
```

4. Elvin for Group Communication

Suppose we have the following message for a group-request message (which is not in the ACL Communicative Act Library but invented ad hoc here):

```
(group-request
  :sender (agent-identifier :name t)
  :receiver (all ?j
              (student-in-class
                 (agent-identifier :name ?j))
            )
  :intended-actor-cdn (done-homework ?j)
  :content "(action ?j raise_hand)"
)
```

The sender is t and the intended receivers are all the students (agents) in the class. The intended actors (who are requested to raise their hand) are not only students in the class but those who have done their homework. How can we implement this using Elvin? We discuss two ways below.

4.1. proxy message with embedded inform message

One way to implement this is as follows. We can translate the above group-request message into the following proxy notification message to Elvin:

```
(proxy
  :sender (agent-identifier :name t)
  :receiver (agent-identifier :name es)
  :content
    "( (all ?x (and
                 (subscribed
                    (agent-identifier :name ?x)
                    (subexp ?e)
                 )
                 (matches (subexp ?e)
                          (notification n)
                 )
               )
       (action (agent-identifier :name es)
         (inform
           :sender (agent-identifier :name es)
           :content \"(notification n)\"
         )
         true
       )
    )"
)
```

where the notification n is

```
LOCATION: "in class"
TASK-CDN: "done homework"
ACTION: "raise_hand"
```

A matching subscription expression e is as follows:

```
(LOCATION == "in class")
```

If an agent has subscribed with this expression, it will receive the notification, and on receiving the notification, it processes the TASK-CDN attribute and checks if it has done its homework, before deciding on its action. Alternatively, an agent who has done its homework might have subscribed to Elvin with the following expression:

```
(LOCATION == "in class") &&
        (TASK-CDN == "done homework")
```

and so, have left it to Elvin to check the intended actor's condition. With this subscription, the agent will receive notifications from t intended for those in class *and* have done their homework.

However, this way embeds the content of the group-request message in the notification. We rather have the Elvin subscription-notification matching be used only to resolve the set of receivers.

4.2. `proxy` message with embedded `request-when` message

An alternative way is to use the following `proxy` notification message to Elvin, where the notification is as before but used only to find the agents with matching subscriptions, and to each agent that matches the subscription, a `request-when` message is sent instead of an `inform` message. The contents of the `request-when` message is a (<action> <proposition>) pair where the action is `raise_hand` and the action is done when the proposition `done-homework` is true, i.e. the message requests that the agent (student) raises its hand if it has done its homework.

```
(proxy
  :sender (agent-identifier :name t)
  :receiver (agent-identifier :name es)
  :content
    "( (all ?x (and
                 (subscribed
                    (agent-identifier :name ?x)
                    (subexp ?e)
                 )
                 (matches (subexp ?e)
                          (notification n)
                 )
               )
       (action (agent-identifier :name es)
         (request-when
           :sender (agent-identifier :name es)
           :content \"(
             (action
                (agent-identifier :name ?x)
                raise_hand)
             (done-homework ?x)
           )\"
         )
         true
       )
    )"
)
```

We emphasize the point that the sender of the above `proxy` message does not know who the recipients will be or the actual identities of the intended actors, as consistent with the type of group communication mentioned in [3].

5. Conclusions and Future Work

We have presented a set of communicative acts for mobile agents to interact meaningfully with the Elvin service, namely, (1) `request-whenever` for subscribing to Elvin, (2) `cancel` for unsubscribing from Elvin, (3) `proxy` for notifications from producers to the Elvin server, and (4) `inform` for notifications from the Elvin server to subscribers (or consumers). We contend that such a mapping can be used for event notification systems in general, and not just Elvin.

6. References

[1] DSTC. The Elvin Subscription Language. http://elvin.dstc.edu.au/doc/esl4.html

[2] FIPA. FIPA Communicative Act Library Specification, 2002. http://www.fipa.org/specs/fipa00037/

[3] Kumar, S., Huber, M.J., McGee, D.R., Cohen, P.R., and Levesque, H.J. Semantics of Agent Communication Languages for Group Interaction. *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*, AAAI Press, Austin, Texas, July 30-August 3, 2000, pp 42-47.

[4] Searle, J. R. *Speech Acts--An Essay in the Philosophy of Language*. Cambridge University Press, London, 1969.

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