Coordination Middleware Supporting Rapid Deployment of Ad Hoc Mobile Systems

Radu Handorean, Jamie Payton, Christine Julien, and Gruia-Catalin Roman
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{raduh, payton, julien, roman}@cse.wustl.edu

Abstract

This paper addresses the design and implementation of thin coordination veneers for use in the development of applications over ad hoc wireless networks. A coordination veneer is defined as an adaptation layer that customizes a general-purpose coordination middleware to a particular domain with minimal development effort. This technique allows developers to build highly-tailored coordination models while leveraging established models and middleware. We present three such veneers, the coordination models they embody, and the manner in which they were implemented. The LIME *middleware, which supplies tuple space based coordination in the ad hoc environment, serves as the implementation base for our veneers. These veneers cover diverse areas in ad hoc mobility: service discovery and provision, event registration and distribution, and secure tuple space access.*

1 Introduction

The advent of wireless communication has opened a wide range of new opportunities for staying in touch while travelling. Significant investments have been made in maintaining access to the Internet and its resources regardless of location. At times, access may even be tailored to the current location. A related, complementary trend involves the emergence of ad hoc networks. Even though they rely on the same wireless technology, ad hoc networks remove the reliance on base stations and allow devices to communicate with each other whenever in communication range.

Rapid and dependable deployment of applications over ad hoc networks proves difficult in the presence of mobility. While the intrinsic complexity of the task cannot be avoided, the programmer can be protected from it by employing appropriately designed middleware. In this paper we advance the proposition that coordination middleware can play an important role in simplifying the development of applications in ad hoc environments.

The key advantages of a coordination-based strategy are the strong emphasis on decoupling among components and the delegation of the communication details to the underlying middleware. Coordination models (à la Linda [6]) were originally developed to support decoupled interactions among concurrent processes by offering a small set of primitives (**in**, **rd**, and **out**) for content-based access to a global, persistent, shared data structure (a tuple space). More recent work has extended this approach to support inter-agent communication in fixed networks (e.g., MARS [2], Jini [5], TSpaces [11], etc.) and even host-to-host coordination in mobile ad hoc networks (e.g., LIME [13]). In this paper we take this technology one step further by considering the software engineering implications of providing coordination middleware specialized for a particular class of applications. Our objective is to demonstrate the feasibility of constructing such specialized coordination middleware with a relatively small investment in new software development.

We begin with the assumption that an application is structured in terms of code fragments distributed over hosts which communicate via wireless transmitters. Code fragments can migrate among hosts when connectivity is available and will be treated as mobile agents. An agent becomes both a unit of mobility and a unit of modularity. Different applications targeted to this architecture have diverse coordination needs. We consider three such classes of applications. In each case, we propose a coordination model tailored to the particular setting. The three models are specialized for ad hoc networking and exhibit diverse coordination styles: service provision, event-based notification, and secure tuple space sharing. Despite diversity, we will show that they can be constructed with a relatively small amount of effort as thin adaptation layers (called coordination veneers) over a common coordination middleware supporting

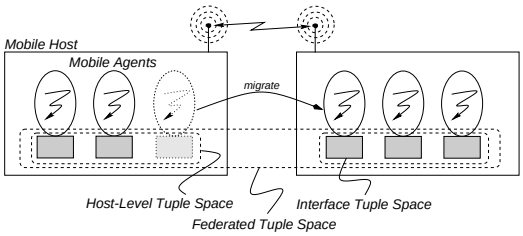

Figure 1. Transiently shared tuple spaces encompass physical and logical mobility.

transient sharing of tuple spaces in ad hoc networks (LIME).

The remainder of this paper is organized as follows. Section 2 gives a brief description of the LIME middleware that the coordination veneers build upon. Sections 3, 4, and 5 detail the models and implementations of the three styles of interaction. Finally, Section 6 provides conclusions.

2 A Review of Lime

The LIME middleware supports the development of applications exhibiting physical mobility of hosts, logical mobility of agents, or both. LIME adopts a coordination perspective inspired by work on the Linda model.

Transparent Context Maintenance. The model underlying LIME breaks the Linda tuple space into many tuple spaces, each permanently associated to a mobile unit, and by introducing rules for transient sharing of individual tuple spaces based on connectivity. Hence, the content perceived through the tuple space changes dynamically in response to changes in the set of co-located mobile units. Access to the tuple space takes place using the Linda primitives (e.g., **in**, **rd**, **out**). LIME offers probe variants of the traditional blocking operations (e.g., **inp**, **rdp**), and group operations (e.g., **outg**, **ing**, **rdg**, **rdgp**, and **ingp**). While the original calls return a matching tuple (if available) or null otherwise (if nonblocking), the group operations return all matching tuples (or null if none available). LIME also allows for polymorphic tuple matching, where a field in a template matches the respective field of a tuple if the latter contains an object of the same type or of a subtype of the one specified in the template. Figure 1 depicts the LIME model. Mobile agents are the only active components; mobile hosts are roaming containers supporting agents.

Controlling Context-Awareness. LIME provides fine-grained control over the context perceived by the mobile unit by extending Linda operations with tuple location parameters that define projections of the transiently shared tuple space based on agent identities or host identities.

The read-only `LimeSystemTupleSpace` tuple space provides awareness of the system configuration. It contains information about the agents, hosts and tuple spaces in the system. Standard tuple space operations allow access to this information.

Reacting to Changes. LIME extends the basic Linda tuple space with a reaction $\mathcal{R}(s, p)$, defined by a code fragment s that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. After each operation on the tuple space, LIME nondeterministically selects a reaction and compares the pattern p against the tuple space contents. If a matching tuple is found, s is executed, otherwise the reaction is a skip. This selection and execution proceeds until no reactions are enabled, and normal processing resumes. Blocking operations are not allowed in s, as they might prevent the program from reaching fixed point.

Reactions are annotated with locations that restrict the locality of their execution. These kinds of reactions, called *strong reactions*, must always be restricted to the local host or agent. LIME also provides a notion of *weak reaction* in which the execution of s does not happen atomically with the detection of a tuple matching p; instead, it is guaranteed to take place eventually if connectivity is preserved.

Maintaining Group Membership in Highly Dynamic Contexts. LIME protects applications from the complexity associated with sudden disconnection by using location information. The concept of safe distance [19] helps preserve the consistency of the system by predicting disconnections. When a host approaches a group, it is allowed to *engage* with the group only after it comes within safe distance of some member of the group. Once the safe distance is exceeded, an automatic *disengagement* protocol is triggered and the group is split, ensuring that no messages between group members are lost and that messages are sent and received in the same configuration.

Software Distribution. LIME is available under a GNU's LGPL open source license. Source code and development notes may be obtained from `lime.sourceforge.net`.

3 Service Provision

In the client-server model, which continues to dominate distributed computing, the client knows the name of the server that supports the service it needs, has the code necessary to access the server, and knows the communication protocol the server expects. More recent strategies allow one to advertise services, to lookup services and to access them without explicit knowledge of the network structure and communication details.

3.1 Service Provision Models

The service model is composed of three components: services, clients, and a discovery technology. Services provide needed functionality that clients use. The discovery

process enables clients to find and use services advertising particular capabilities. After a successful lookup, a client receives a piece of code implementing the service or facilitating communication to the server offering the service.

In service provision models, clients may discover services at runtime and use them through proxies the services provide. A proxy hides the network from the client by offering a high-level interface, for using the service, while the proxy's interaction with the server remains unknown to the client. Services are advertised by publishing a profile containing attributes and capabilities useful to a client when searching for a service. Servers aggregate these published profiles into a service registry that clients can search using templates generated according to their momentary needs.

Different implementations of the service model currently exist. Our work is significantly influenced by Sun Microsystems's Jini [5, 14] model, which uses service registry lookup tables managed by special services called lookup services. A Jini community cannot work without at least one lookup service, even if services and potential users reside on the same physical host. Other approaches include IETF's Service Location Protocol [9]; Microsoft's Universal Plug'n'Play [12], which uses the Simple Service Discovery Protocol [7]; and the Salutation project [15].

3.2 A Service Provision Veneer

All these models assume a more or less stable network. Mobile ad hoc networks are opportunistically formed structures that change in response to the movement of physically mobile hosts running potentially mobile code. The service model needs to adapt to the ad hoc environment. For example, if the node hosting the service registry suddenly becomes unavailable, the advertising and lookup of services are paralyzed even if the pair of nodes representing a service and a potential client remains connected (Figure 2).

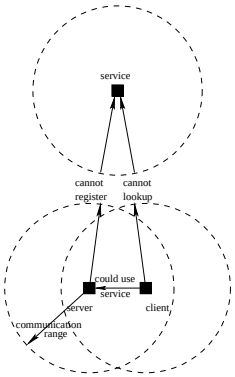

Figure 2. The client could use the service but not discover it since the service registry is not accessible.

In our model, described in more detail in [10], services continue to be advertised by publishing a profile that contains the capabilities of the service and attributes describing these capabilities. With its profile, a service provides a service proxy which represents the service locally to the client. If a service profile satisfying all client requirements is available, the service proxy is returned to the client. The client uses the proxy to interact with the service as if it were local.

In our model, an advertisement for a specific service can be discovered if and only if the service is available. We accomplish this by making sure that discovery and accessibility of remote servers is scoped by host connectivity. The result is a federated service registry containing the union of all local tuple spaces in the connected ad hoc network and atomically updated as connectivity changes. Thus, when the host of the service becomes unreachable (i.e., disconnected), the local repository atomically becomes unavailable as well, and the service can no longer be discovered. The approach eliminates the need for a centralized directory for registration and lookup and it guarantees that two hosts within communication range can exchange services. Finally it prevents a client from discovering a service that is no longer available at the time of the lookup. Figure 3 depicts the typical usage of the distributed service registry.

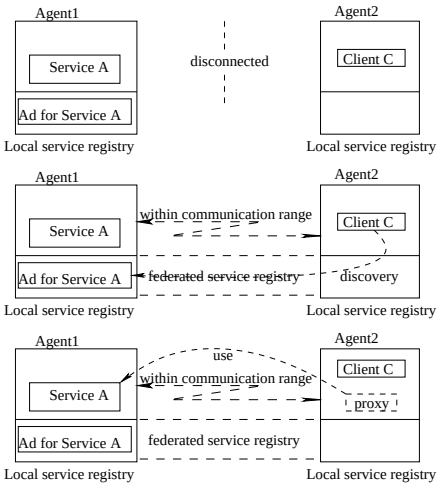

Figure 3. Local service registry sharing and proxy service utilization.

3.3 Implementation

Each service is represented by a profile stored in a tuple along with the proxy object. When the service is registered, the system assigns to it a globally unique service id. This id represents the service as long as it is available and can be used for rediscovery. Service attributes in the profile quantify the capabilities of the service (e.g., "color" and "laser" can be attributes for a service advertising the "print" capability). The client may use attributes when searching for

services to filter the results. The proxy object is a piece of code that can either fully implement the service or provide an interface to a remote service provider. The latter situation is encountered when the service needs a specific piece of hardware to do the job (e.g., a printer), or some resource that cannot migrate to the client.

To search for a service, a client calls the method *lookup(ServiceTemplate sTempl)*, where *ServiceTemplate* contains the service id, the list of needed capabilities, and a list of required attributes of the capabilities. The call returns the proxy object if a service was found or null otherwise.

While the proxy hides the network from the client, the proxy must know where the server is located. In the presence of mobility, the location information may change upon migration of the service. For example, if the agent providing the service moves to another host, the IP address (physical location) changes, but the port number (logical location) may not. While mobility causes physical location to change, this logical address is not likely to change. Since the physical location is unique for all servers run by each mobile agent, but the logical address is specific to each server and does not change, we publish them separately. The physical location is published along with the agent's id in a special tuple space, called the `ServiceLocationTupleSpace`. This tuple space contains one location tuple for each agent, and the content is updated upon agent migration. The tuple space used for advertisements `LookupServiceTupleSpace` contains tuples that represent services, including their logical addresses. This way, an agent needs to update only the tuple in `ServiceLocationTupleSpace` when it migrates, regardless of the number of services it provides. The logical address is part of the tuple that contains the advertisement of the service. Upon migration, these tuples will follow the agent automatically and remain unchanged.

Using the `LookupServiceTupleSpace` name for all local lookup tuple spaces, allows us to take advantage of LIME's transient sharing of tuple spaces. Since host engagement and disengagement are atomic operations, each agent sees a consistent set of services available across the ad hoc network.

In the template used to query the federated tuple space, the client can request a service with a specific id, services that have certain attributes, services that implement certain interfaces, or a combination of the above. A tuple is considered to match the client's requirements if the service it advertises has all the properties the client demands. The attributes and capabilities specified in a template must be subsets of the attributes and capabilities published for the matching service. Attributes compared should match as values, while the capabilities are allowed to match according to types in a polymorphic fashion.

Mobile agents run the clients and the services. An agent running a client or an agent providing a service may migrate to a new host. With tuple space based communication, no special measures are required to resume collaboration between the client and the server when migration occurs. The tuple spaces are automatically transferred to the new location, and continue to be uniformly accessed, since the location does not influence the process of tuple retrieval.

If the client code migrates, a private socket protocol between the proxy and its server must reopen the communication channel with the server, using the same location information. If the server code migrates, its physical location tuple must be updated. The clients will need to reconnect to the server using the new location information.

Agent migration in LIME is supported via μCode [16]. The implementation preserves the memory state, but not the control state. This means that at its destination, the agent restarts execution with the memory initialized to the content present when the migration was triggered. This initialization includes the re-registration of the services. Having the memory content preserved helps implement a resume behavior. That is, it can only perform those actions from the registration that are absolutely needed (e.g., it can only update the location tuple). This also allows the client and the server to resume the communication from a certain point without restarting the entire task.

4 Event Distribution

In an event distribution model, a component can generate (publish) event notifications and can specify the set of event notifications it receives using subscriptions. A component may also remove a subscription for an event. Components subscribe to event notifications and publish event notifications through an event dispatching service. Recent work suggests the need to adapt the event distribution model for use in mobile environments [4, 3] by using each component as a publisher, a subscriber, and an event dispatcher.

4.1 An Event Distribution Veneer

In our model, agents utilize an event repository to achieve communication. The event repository is represented as a tuple space. Each agent in the system is associated with its own local tuple space, which is transiently shared. The shared tuple space is used as the event dispatching service. Agents publish event notifications to a tuple space and subscriptions are made on the same tuple space.

An agent generates an event notification using *Publish(Tuple notification)* where *notification* is a set of fields that defines an event notification. The event notification is placed into the tuple space. The notification is made available to agents connected to the originator of the notification through transient sharing of local tuple spaces.

An agent registers for an event to receive notification of its occurrence using *Subscribe(ITuple template, LimeEventListener listener)*. The parameter *template* is a set of fields that specifies a pattern for event notifications. This template is used in pattern matching with published event notifications. The *listener* is a handle to a callback function that executes if an event notification that matches the specified template is generated. Multiple subscriptions may be based on the same event notification pattern, each specifying a different callback function to be executed.

Unsubscribe operations are used to remove subscriptions for event notifications. *Unsubscribe(ITuple template, LimeEventListener listener)* is used to remove a specific subscription. To ensure the correct subscription is removed, both the event notification pattern and the handle to code used in the original subscription are provided as parameters. The *UnsubscribeAll(ITuple template)* operation removes *all* subscriptions previously registered upon the event notification pattern provided as a parameter.

In unsubscribe operations, the pattern provided must exactly match a pattern in a registered subscription, otherwise the unsubscription is ignored and the user is notified.

At times it may be beneficial to limit the scope of notifications. We include a *Subscribe(location source, ITuple template, LimeEventListener listener)* operation which allows an agent to subscribe for notifications generated by a specific agent, identified by the parameter `source`.

4.2 Implementation

Communication is achieved using transiently shared tuple spaces. In LIME, tuple space sharing is possible only when agents have identically named tuple spaces. Therefore, each agent is associated with a tuple space called the `EventTupleSpace`.

Subscribe(ITuple template, LimeEventListener listener) is implemented using LIME reactions. A reaction is defined by a pattern that describes a tuple, and a callback function. We use the weak reaction provided in LIME, in which atomicity requirements of reactions are relaxed. This guarantees that upon appearance of a notification in the `EventTupleSpace` matching the provided pattern, the code specified by *listener* will be eventually be executed, as long as connectivity between the publishing agent and the subscribing agent is preserved. These reactions (subscriptions) are stored in a hash table, using *template* as a key.

To allow a reaction to fire for every new occurrence of an event notification tuple, we define a reaction as occurring once per tuple. A once per tuple reaction in LIME fires only for tuples that match the pattern specified by the reaction and that have not previously triggered the reaction. To support such reactions, agents perform bookkeeping operations as reactions are fired to record the id of the tuple that triggered the reaction. Each time a tuple is placed into the tuple space, the bookkeeping information is checked before a reaction is actually fired.

The unsubscribe operations are realized by deregistering the reactions. In the *Unsubscribe(ITuple template, LimeEventListener listener)* operation, the programmer specifies the exact subscription that should be removed by providing the same parameters used to register the subscription. The subscription hash table is searched to determine if such an entry exists. If so, then it is removed using the LIME *removeWeakReaction* operation. In the *UnsubscribeAll(ITuple template)* operation, the programmer specifies the pattern for which it no longer is interested in receiving notifications. The subscription hash table is searched to determine if subscriptions for that event notification pattern exist. If so, then all reactions in the hash table that were registered on that event notification pattern are removed.

An agent must explicitly unsubscribe for all related subscriptions. Otherwise, it is possible for an agent to still receive notifications in which it is no longer interested because of overlapping subscriptions.

Publish(Tuple notification) uses the LIME **out** primitive to place a notification in the `EventTupleSpace`. To clean up the tuple space, when publishing an event notification, the LIME **out** operation which places the tuple in the `EventTupleSpace` is immediately followed by a LIME **in** operation to remove the event notification from the tuple space. It might seem that removing the notification in this way could prevent the delivery of the notifications to subscribers. Since reactions are used in subscriptions, this is not the case. All reactions are guaranteed to fire upon the appearance of an event notification matching its pattern in the tuple space.

One issue of interest is unannounced disconnection of hosts due to physical movement. If a host becomes disconnected and is later reconnected, the agents on that host will only receive event notifications generated since the time of reconnection. The event distribution veneer currently does not support queueing of events upon unannounced disconnection of hosts for two reasons. First, the underlying model for this veneer, LIME, limits the firing of reactions to connected agents. Second, in some cases, event notifications should not be received while a host or agent is disconnected.

Another point of interest is the subscription mechanism. Our event distribution model provides a form of content based subscription. However, unlike other comparable event distribution models [3, 1, 22], our model does not currently support matching based on predicate evaluation. While the LIME pattern matching mechanism has recently been extended to support such evaluation, this method of pattern matching has not yet been incorporated into use with the event veneer.

5 Secure Transparent Data Sharing

Mobile agent systems bring radical changes in application design. In particular, security concerns come to the forefront in highly dynamic mobile environments. When considering coordination among mobile agents, security concerns can be grouped into three categories: protecting hosts from malicious agents, protecting agents from malicious hosts, and protecting the integrity of the data.

5.1 Security in Mobile Agent Systems

Tuple space based infrastructures are well suited for mobile agent coordination and communication. The original Linda model and many of its successors, however, do not address security issues. Without protection, applications developed on top of a tuple space infrastructure remain of purely academic interest. Several systems attempted to add security to tuple space coordination of mobile agents (e.g., KLAIM [17]). Others address the problem of protecting hosts from malicious agents. The D'Agents [8] system uses public-key cryptography to authenticate incoming agents. The problem of protecting agents from malicious hosts led to agents computing with encrypted functions [20, 18].

5.2 A Secure Tuple Space Veneer

The LIME model supports multiple tuple spaces but offers no security mechanisms. The secure tuple space veneer compensates for this by providing password protected tuple space access. In the LIME implementation, simply knowing the name of a tuple space allows an agent to access it (i.e., read information, remove information, write information). Since LIME allows for sharing tuple spaces with the same name, accessing a federated tuple space is as easy as accessing a local one. Moreover, the use of polymorphism in pattern matching makes tampering with the contents of a tuple space particularly easy. This veneer overcomes these problems by requiring password authentication for use of special secure tuple spaces. The veneer allows for the existence of both protected and unprotected tuple spaces.

`LimeSystemTupleSpace` is a special LIME tuple space which contains, among other things, the names of all tuple spaces. Any agent can read from this tuple space, and therefore, any agent can discover tuple spaces and attempt to access them. Our solution protects the tuple spaces by granting an agent access only after it provides a correct password. The real tuple space name results from encrypting the tuple space name specified by the programmer with the provided password. Once an agent obtains a handle to the tuple space, the password does not have to be used anymore during normal interaction with it. LIME limits the access to the tuple space only to those agents that created the tuple space locally and then shared it with others. Thus, it is impossible for an agent to use a handle obtained from another agent (even if the latter obtained the handle correctly).

The execution of a method can involve local and/or remote (federated) tuple space access. If the call involves remote execution, the parameters will be sent across the network encrypted with the password provided. If the password is correct, the parameters will be decrypted correctly, and the remote host will be able to execute the requested command. Backward compatibility is preserved by allowing the use of unprotected tuple spaces.

The encrypted names of tuple spaces are still visible in the `LimeSystemTupleSpace`. However, they cannot be used inappropriately. If a tuple space is password protected, then the encrypted name from the `LimeSystemTupleSpace` can only be generated from the clear name used in conjunction with the correct password. An attempt to create a tuple space with a name identical to a mangled name from the `LimeSystemTupleSpace` but without a password will generate an exception. Thus, copying the name of a (protected) tuple space is no longer useful.

5.3 Implementation

The interface provided by the secure coordination veneer is almost identical to the interface that LIME provides for an agent. The difference is that tuple spaces are now created using the SecureLimeTupleSpace class. As mentioned above once the agent has the handle to the tuple space, it is free to access it unrestrictedly. Since this handle is not transferrable, it is not necessary to ask for the password in every single interaction with the tuple space. Therefore all other tuple space operations remain unchanged.

Before creating the tuple space, the name (clear if the tuple space is not password protected or encrypted otherwise) is prefixed with the letter "U" if unprotected or "S" if it is a protected tuple space. This step ensures that names copied from `LimeSystemTupleSpace` cannot be used directly.

Whenever a new secure tuple space is created, a new entry of the form [encrypted name, password] is added to a **SecurityTable** in the LIME server. The system uses this table to decrypt incoming messages, and to encrypt outgoing messages. Unless otherwise indicated by location parameters, all calls operate over the entire federated tuple space, including the local tuple space and tuple spaces with the same name residing on other hosts.

The implementation is based on the interceptor pattern [21]. When an agent executes a call that leads to a message being pass to another host, an interceptor catches it and performs the encryption/decryption before forwarding it to the network or LIME system, respectively. The encryption

algorithm used is 3DES. Figure 4 shows how the interceptors function to secure the tuple space communication.

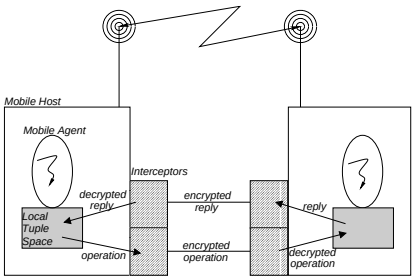

Figure 4. Interceptors catch messages and encrypt them before sending and decrypt after receiving.

The distribution of the passwords remains an open issue. This paper does not address this problem. If two agents want to interact using a protected tuple space, they need the name of the tuple space and the password a priori.

6 Conclusions

The paper describes three coordination veneers that support development of applications over ad hoc networks. Our veneers can be used both for applications that require specific types of coordination, and in combination. All veneers have been implemented and will soon be publicly available.

The idea of creating specialized coordination models and middleware for particular applications is new. The approach holds the promise for major simplifications in the development of software systems over ad hoc networks. Equally important is that all these veneers were constructed with minimal effort over an existing coordination middleware (LIME). We have been able to demonstrate the feasibility of employing specialized coordination middleware in software development while, also offering additional evidence regarding the expressive power of LIME and its model.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant No.CCR-9970939. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of the 19^{th} Int'l Conf. on Distributed Computing Systems*, pages 262–272, 1999.

[2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *Internet Computing*, 4(4):26–35, 2000.

[3] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 19(3):332–383, 2001.

[4] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. on Software Engineering*, 27(9):827–850, September 2001.

[5] K. Edwards. *Core JINI*. Prentice Hall, 1999.

[6] D. Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*, 7(1):80–112, January 1985.

[7] Y. Goland, T. Cai, P. Leach, Y. Gu, Microsoft Corporation, S. Albright, and Hewlett-Packard Company. Simple service discovery protocol/1.0: Operating without an arbiter. http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt.

[8] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 154–187. Springer-Verlag, 1998.

[9] E. Guttman. Service location protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, 4(3):71–80, July-August 1999.

[10] R. Handorean and G. C. Roman. Service provision in ad hoc networks. In *Coordination Models and Languages*, volume 2315 of *LNCS*, pages 207–219. Springer-Verlag, 2002.

[11] IBM. T Spaces. http://www.almaden.ibm.com/cs/TSpaces/.

[12] Microsoft Corporation. Universal plug and play forum. http://www.upnp.org, 2001.

[13] A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proc. of the 21^{st} Int'l Conf. on Distributed Computing Systems*, pages 524–533, April 2001.

[14] J. Newmarch. *Guide to Jini Technologies*. http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml.

[15] B. Pascoe. Salutation architectures and the newly defined service discovery protocols from Microsoft and Sun. http://www.salutation.org/whitepaper/Jini-UPnP.PDF.

[16] G. Picco. μCode: A lightweight and flexible mobile code toolkit. In *Proc. of the 2^{nd} Int'l Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 160–171. Springer-Verlag, September 1998.

[17] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *Software Engineering*, 24(5):315–330, 1998.

[18] J. Riordan and B.Schneier. Environmental key generation towards clueless agents. In *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 15–24. Springer-Verlag, 1998.

[19] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Proc. of the 23rd Int'l Conf. in Software Engineering*, 2001.

[20] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agent Security*, LNCS, pages 44–60. Springer-Verlag, 1998.

[21] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern Oriented Software Architecture*, volume 2. John Wiley & Sons, Ltd., 1999.

[22] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proc. of the Australian UNIX and Open Systems User Group Conf. (AUUG97)*, September 1997.