Report Number: WUCSE-2002-33

2002-09-12

Rapid Deployment of Coordination Middleware Supporting Ad Hoc Mobile Systems

Gruia-Catalin Roman, Jamie Payton, Radu Handorean, and Christine Julien

This paper is concerned with the design and implementation of think coordination veneers for use in the development of applications over ad hoc wireless networks. A coordination veneer is defined as an adaption layer that customizes a general-purpose coordination middleware to a particular application domain with minimal development effort. This technique allows developers to build highly-tailored coordination models while leveraging off 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... Read complete abstract on page 2.

Rapid Deployment of Coordination Middleware Supporting Ad Hoc Mobile Systems

Gruia-Catalin Roman, Jamie Payton, Radu Handorean, and Christine Julien

Complete Abstract:

This paper is concerned with the design and implementation of think coordination veneers for use in the development of applications over ad hoc wireless networks. A coordination veneer is defined as an adaption layer that customizes a general-purpose coordination middleware to a particular application domain with minimal development effort. This technique allows developers to build highly-tailored coordination models while leveraging off 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 underlying implementation base for our veneers. These veneers cover diverse application areas in ad hoc mobility: service discovery and provision, event registration and distribution, and secure tuple space access.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2002-33

Rapid Deployment of Coordination Middleware Supporting Ad Hoc Mobile Systems

Authors: G.C. Roman; J Payton; R. Handorean; C. Julien;

September 12, 2002

Abstract: This paper is concerned with 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 application domain with minimal development effort. This technique allows developers to build highly-tailored coordination models while leveraging off 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 underlying implementation base for our veneers. These veneers cover diverse application areas in ad hoc mobility: service discovery and provision, event registration and distribution, and secure tuple space access.

Rapid Deployment of Coordination Middleware Supporting Ad Hoc Mobile Systems

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

ABSTRACT

This paper is concerned with 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 application domain with minimal development effort. This technique allows developers to build highly-tailored coordination models while leveraging off 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 underlying implementation base for our veneers. These veneers cover diverse application 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 traveling. Significant investments have been made in maintaining access to the Internet and its resources regardless of one's location. At times, access may even be tailored to one's current location. A related trend that complements these developments 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 they happen to be in communication range. A small niche market at this point, ad hoc networks and applications are expected to grow rapidly in importance because they meet the specific requirements of a number of important application domains. Applications in settings such as disaster response, planetary exploration, underground infrastructure maintenance, mine inspection, etc., cannot reasonably expect wired network support, yet the need to facilitate collaborative work prevails. Additionally, applications may often include safety critical components. One such example is underground repair work on high voltage power lines in which team members must cooperate very accurately to avoid accidents. Similarly, exploring a partially collapsed building requires all members of the team to be continuously in touch and to follow specific cooperative inspection protocols. In other settings, Internet access may be available but may not be the best medium for peer-to-peer coordination. A driver may carry a PDA, which can interact more economically with devices on the vehicle via a direct wireless connection. On the factory floor, robots and built-in sensors may find it more effective to communicate directly with each other over short distances.

The development of these new kinds of applications poses novel challenges for today's software engineers. Rapid and dependable deployment of applications over ad hoc networks proves difficult in the presence of mobility, variability in motion profiles, unpredictable and frequent disconnection, limited resources associated with small devices, and the open nature of the resulting environment. 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 degree to which an application can delegate the communication details to the underlying middleware. Coordination models (à la Linda [1]) 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 [3], TSpaces [4], etc.) and even host-to-host coordination in mobile ad hoc networks (e.g., LIME [5]). 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 start our investigation with the assumption that an application is structured in terms of code fragments distributed over hosts which communicate via wireless transmitters. The hosts can move in some physical space in and out of their respective communication ranges. Code fragments can migrate among hosts when connectivity is available and, for this reason, will be treated as mobile agents whether or not they qualify in the strictest sense of the word. For us, an agent becomes both a unit of mobility and a unit of modularity. Different applications targeted to this archi-

tecture 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 (henceforth called coordination veneers) over a common coordination middleware supporting 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: service provision, event distribution, and secure transparent data sharing, respectively. Finally, Section 6 provides some 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. The context for computation, represented in Linda by a globally accessible, persistent tuple space, is represented in LIME by transient sharing of tuple spaces carried by each individual mobile unit. LIME also extends Linda tuple spaces with a notion of location and with the ability to react to a given state.

Transparent Context Maintenance. The model underlying LIME accomplishes the shift from a fixed context to a dynamically changing one by breaking the Linda tuple space into many tuple spaces, each permanently associated to a mobile unit, and by introducing rules for transient sharing of the individual tuple spaces based on connectivity. From the perspective of a mobile unit, the only way to access the global context is through an *interface tuple space* (ITS), permanently and exclusively attached to the unit itself. The ITS contains tuples the mobile unit is willing to make available to other units, that are co-located with the unit itself. This represents the only context accessible to the unit when it is alone. This tuple space is *transiently shared* with ITSs belonging to mobile units that are connected. Hence, the content perceived through the ITS changes dynamically in response to changes in the set of co-located mobile units. Access to the ITS takes place using the Linda primitives (e.g., **in**, **rd**, **out**), whose semantics are basically unaffected. LIME offers an extension to this synchronous communication by providing probe variants of the traditional blocking operations (e.g., **inp**, **rdp**). LIME also offers several other extensions of the traditional Linda model, designed to handle groups of tuples (e.g., **ing**, **rdg** and **outg**) as well as their nonblocking variants (where applicable) (e.g., **ingp** and **rdgp**). 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 and the call is nonblocking).

Another extension to the traditional Linda model is a new pattern matching mechanism. In LIME we allow for polymorphic tuple matching, i.e., a field in the template provided for pattern matching will match 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. This allows for wild cards in pattern matching (e.g., the Object class in Java will match anything) and for the use of interfaces to retrieve objects that implement them.

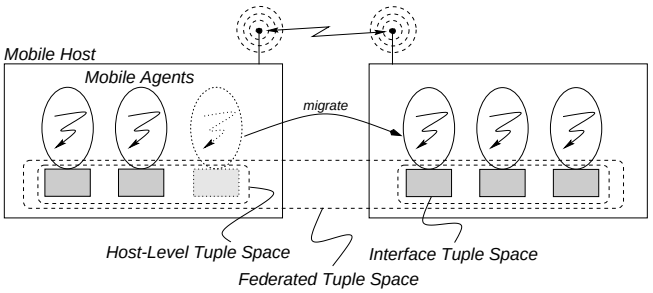

Figure 1: Transiently shared tuple spaces encompass physical and logical mobility.

Encompassing Physical and Logical Mobility. In an ad hoc network, LIME mobile hosts are connected when the distance between them allows communication. Mobile agents are connected when they are co-located on the same host, or they reside on hosts that are connected. Creation and termination of mobile agents is a special case of connection and disconnection, respectively. Figure 1 depicts the LIME model. Mobile agents are the only active components; mobile hosts are mainly roaming containers providing connectivity and execution support for agents. In other words, mobile agents are the only components that carry a "concrete" tuple space with them. Multiple agents' tuple spaces that are transiently shared across hosts form a *federated tuple space*.

Controlling Context Awareness. LIME fosters a style of coordination that reduces the details of distribution and mobility to changes in what is perceived to be a local tuple space. This view is powerful as it relieves the designer from specifically addressing the changes in configuration, but some mobile applications need to explicitly address the distributed nature of the data for performance or optimization reasons. LIME provides such 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. LIME expresses tuple location parameters in terms of agent identifiers or host identifiers. Both can be used both to place tuples at a particular agent location or to restrict queries to specific agents or hosts.

The read-only `LimeSystemTupleSpace` tuple space provides awareness of the system configuration. Its tuples contain information about the mobile units present in the community, and their relationship, e.g., which tuple spaces they are sharing or, for mobile agents, which hosts they reside on. Standard tuple space operations on the `LimeSystemTupleSpace` tuple space allow an agent to respond to the arrival and departure of other agents and hosts.

Reacting to Changes in Context. Mobility enables a highly dynamic environment, where reaction to changes constitutes a major fraction of the application design. Therefore, LIME extends the basic Linda tuple space with the notion of a *reaction*. A reaction $\mathcal{R}(s, p)$ is 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 there are no reactions enabled, and normal processing resumes. Thus, reactions are executed as if they belonged to a separate reactive program which runs to fixed point after each non-reactive statement. Blocking operations are not allowed in s, as they might prevent the program from reaching fixed point.

Actually, the full form of a reaction is annotated with locations that restrict the locality of their execution. Moreover, these kinds of reactions, called *strong reactions*, are not allowed over the entire federated tuple space; they must always be restricted to a host or agent. Otherwise, maintaining the requirements of atomicity and serialization imposed by strong reactive statements would require a distributed transaction encompassing multiple hosts for every tuple space operation. LIME also provides a notion of *weak reaction*. Processing of a weak reaction proceeds as in the case of strong reactions, except that 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. Environments characterized by frequent disconnections can seriously affect system performance and maintenance. For this reason, the initial version of LIME made the simplifying assumption of announced disconnection. This allowed operations to complete and communication to stop before the actual disconnection occurs. The current version of LIME protects applications from the complexity associated with sudden disconnection by using location information in the engagement/disengagement protocol.

The concept of safe distance [6] was introduced to help preserve the consistency of the system by predicting disconnections. Two hosts are considered to be at a safe distance if given the speed of the two hosts, any task in progress is guaranteed to complete before any disconnection can occur. The safe distance is usually a fraction of the range of the wireless transmitter. Once the safe distance is exceeded, an automatic disengagement protocol is triggered and the group is split. The safe distance ensures that no messages between group members are lost and that messages are sent and received in the same configuration. This helps prevent inconsistent states when the actual disconnection occurs. When a host approaches a group, it is allowed to engage the group only after it comes within safe distance of some member of the group. This ensures a clean and consistent group membership management, in case the host decides to move away from the group soon after engagement. (Since ad hoc routing is not included in the current software release, the groups are clusters of fully connected hosts)

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`.

In the remainder of this paper, we reuse the abstractions and terminology from LIME. We refer to any piece of application software as an "agent" and containers for these agents as "hosts." Agents on the same host automatically share tuple spaces. Hosts can move in physical space, while agents can move from host to host. As hosts move within communication range, tuple spaces belonging to agents on connected hosts are logically merged to form the federated tuple space. In the following sections, we show how the design of specialized coordination middleware takes advantage of the rich set of features offered by LIME.

3. SERVICE PROVISION

As the network infrastructure continues to grow, more and more devices are being directly attached to it. All connected entities can provide services, making the network itself a service repository. 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. As a result of a successful lookup, a client may receive a piece of code that actually implements the service or facilitates communication to the server offering the service.

In service provision models, clients may discover services offered by servers 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. This approach enables a great degree of run time flexibility.

Different implementations of the service model currently exist. Sun Microsystems's Jini [3, 7] 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. In IETF's Service Location Protocol [8, 9], directory agents implement the service registry. They store service profiles and service locations but no executable code. The discovery of services involves first locating these directory agents. If no directory agent is available, clients may multicast requests for services, and servers may multicast advertisements for their services. Microsoft proposed Universal Plug'n'Play (UPnP) [10], which uses the Simple Service Discovery Protocol (SSDP) [11]. This protocol also uses centralized directory services, called proxies, for registration and service lookup. Like the Service Location Protocol, if no such proxy is available, SSDP uses multicast to announce new services or to request services. A service advertisement contains a Universal Resource Identifier (URI) [12] that eventually leads to an XML [13] description of the service. A client can access this description only after it discovered the service through a lookup service. The novelty of this model is the auto-configuration capability based on DHCP [14] or AutoIP. The Salutation project [15] also

uses a relatively centralized service registry called Salutation Manager (SLM). There may be several such managers available, but clients and servers can establish contact only via these SLMs. The advantage of this approach is that these SLMs can have different transport protocols underneath, unlike the above-mentioned models which all assume an IP transport layer. To realize this, Salutation uses transport-dependent modules, called Transport Managers that broadcast internally, helping SLMs from different transport media interact with each other.

3.2 A Service Provision Veneer

All these models assume a more or less stable network. A high degree of freedom and a fully decentralized architecture can be obtained in mobile ad hoc networks, at the expense of facing significant new challenges. 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 becomes paralyzed even if the pair of nodes representing a service and a potential client remains connected. Figure 2 depicts this scenario.

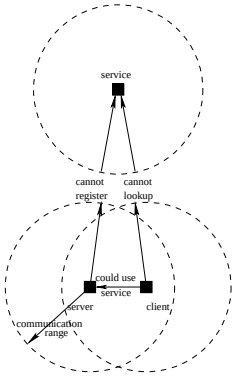

Figure 2: The client could use the service but it cannot discover it since the service registry is not accessible.

In our model, described in more detail in [16], services continue to be advertised by publishing a profile that contains the capabilities of the service and attributes describing these capabilities. Clients use these advertisements to discover and use services properly. The client can use the attributes to decide if the service meets its requirements in terms of quality of service parameters. With its profile, a service provides a service proxy. This proxy will represent the service locally to the client. Clients search for services using a template that defines requirements over the needed service profile. If a service profile satisfying all client requirements is available, the service proxy, as part of the profile, is returned to the client. The client uses the proxy to interact with the service as if it were local.

Maintaining the consistency of data in the service registry is a novel concern our model brings to the surface, appropriately so for such a rapidly changing environment. 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., is disconnected), the local repository atomically becomes unavailable as well, and the service can no longer be discovered. This helps solve several important problems. First, it eliminates the need for a centralized directory for registration and lookup. Second, it guarantees that two hosts within communication range can exchange services. Third, 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.

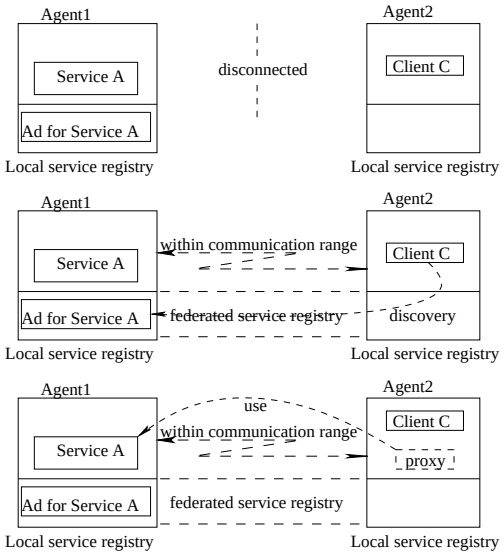

Figure 3: Local service registry sharing and proxy service utilization

3.3 Implementation

LIME offers support for implementing the service model in the mobile ad hoc networking environment. LIME's transient sharing of tuple spaces enables transparent creation of a federated service registry and its atomic update. The interface offered to the programmer is shown in Figure 5 and mimics Jini's API.

3.3.1 Service Representation

Each service is represented by a profile stored as a tuple. This tuple contains a service id, a list of attributes, a list of capabilities, a proxy object, and information about the communication between the proxy object and the server. When the service registers, 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 of the same service. Service attributes 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 may have one of two behaviors: it may fully implement the service with no

remote communication or it may provide an interface to a remote service provider while hiding the details of the communication protocol from the client. The latter situation is encountered when the service needs a specific piece of hardware to execute the job (e.g., a printer), or some resource that cannot migrate to the client. The protocol used by the server and the proxy for their private communication is arbitrary. It can be a well-known protocol (e.g., Java RMI [17]) or a proprietary protocol that is well suited for the application needs.

To search for a service, a client calls the method *lookup(ServiceTemplate sTempl)*, where *ServiceTemplate* contains the serviceID, types (list of needed capabilities), and attributes (list of required attributes of the capabilities). The call returns the proxy object if a service were found or null otherwise. The client can use the proxy object as the local front end for the service.

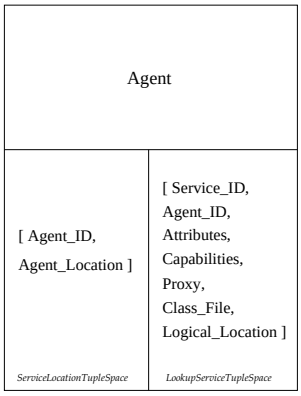

Figure 4: The agent and the two local tuple spaces storing the service profiles and the agent's physical location

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 changes, but the port number may not. Likewise, if the proxy and the server use RMI to communicate, it is very likely for the server to use the same registration string at the new location, even though its IP changed. This observation led us to a design that splits the location information in two parts. One part represents the physical location of the agent running the server, while the other part represents a logical address within the addressing space available at the physical location (e.g., the range of usable ports or the RMI registration strings). 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 (Figure 4). 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 one tuple (the one 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.

3.3.2 Service Access

The tuples describing the services an agent wants to publish are written into a tuple space local to the agent, called the lookup tuple space. By using the same name, *LookupServiceTupleSpace*, for all local lookup tuple spaces, we are able to take advantage of LIME's transient sharing of tuple spaces with the same name. Thus, each agent's lookup tuple space is automatically shared with any co-located agents. In LIME, activities associated with arrival or departure of a host are called engagement and disengagement, respectively. Upon engagement with a new host or group of hosts, this tuple space is shared with all the agents in the community, forming a federated lookup tuple space. Since engagement and disengagement are atomic operations, each agent sees a consistent and up to date set of services available across the ad hoc network.

A client searches for services by querying the federated lookup tuple space as if all information were local by using a template that describes the desired service. In this template, 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. This means the list of capabilities (interfaces) specified by the client in its query template should be a subset of the capabilities advertised for a specific service. In this case, subsetting should be understood to include the polymorphism inherent in the Java programming language. The attributes specified in a template must also be a subset of the attributes published for the matching service. In this latter case, an exact match is performed, i.e., the two attributes compared should match as values, not as types. For example, if a client wants a color (attribute) printer (interface), a service that only specifies printing as a capability without giving details about the quality of printing will not be returned as a possible match for the query.

3.3.3 Service Continuity Upon Migration

Mobile agents run the clients and the services. At some point, an agent running a client or an agent providing a service may decide to 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, if the client and server remain within communication range. 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 agent running the client decides to move, a private socket protocol between the proxy and its server must reopen the communication channel with the server, using the same location information. If RMI is being used for communication, the client will reuse the location information to contact the remote RMI registry and obtain a new proxy object. This is necessary because the RMI implementation embeds the location of both communication ends in

ServiceItem(ServiceID serviceID, java.lang.Object service, Entry[] attrSets)
— prepares a service for registration based on the provided serviceID (null unless this service is being re-registered), service (proxy object), and array of attributes describing the service. The result is used in the call below to register a service.

register(ServiceItem item, long leaseDuration)
— publishes a service specified by the service item for the provided duration. Our implementation preserves the leaseDuration parameter to match the Jini API but does not use it.

ServiceTemplate(ServiceID serviceID, java.lang.Class[] serviceTypes, Entry[] attrSetTemplates)
— prepares a template for matching a desired service based on the serviceID (which can be null), the service types that the client requires the service to implement, and the attributes that the client requires. The result is used to lookup a matching service.

lookup(ServiceTemplate tmpl)
— searches for the service matching the template and returns the proxy object if a service is found and null otherwise.

Figure 5: Service Provision Veneer API

the proxy object generated (i.e., an RMI stub object cannot be transferred and reused on a different host from the one where it was deployed by the RMI infrastructure; the stub is tied to the host where it was first deployed).

If the agent running server code migrates, its physical location tuple must be updated. The clients will need to reconnect to the server using the new location information. In the case of RMI communication, the server also needs to reregister with the new RMI registry at the new location. The client will need to download a new proxy object (RMI stub file) from the RMI registry using the new physical location information.

Agent migration in LIME is supported via μCode [18]. 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.

Successful tests of this veneer have been and continue to be performed. A limitation of the current implementation is that the client must have on its host the class file of any service it requests. Further development of this veneer will include downloading the class file on demand, via the tuple space.

3.4 Application Example

For a typical application of the **service provision veneer** consider the following setting. When a programmer enters a laboratory, the PDA that she/he carries in the pocket joins the ad hoc network already created by the multitude of gadgets in the lab. The agent running on the PDA will obtain proxy objects from different services that control the temperature, the light intensity, musical background, security system, e-mail and so on. As the programmer enters, the agent will collect the proxies and will transmit to their servers the user's preferences. The temperature, light intensity and music volume can be adjusted to the average of the desired values specified by those already in the room. New e-mail messages will be downloaded to the PDA or at least a notification of their availability can be sent. The security system can perform a background verification by comparing the credentials transmitted by the PDA and images taken by video cameras with information stored on a server.

4. EVENT DISTRIBUTION

Interaction between applications across a network is often required to accomplish computing tasks. Though using a client-server model of communication is still a common practice, interest in using an event distribution model to foster communication among distributed applications has dramatically increased in recent years. Event distribution models have become popular, in part, because they hide the complexities involved in facilitating application communication. In event distribution models, communication is asynchronous and anonymous. Systems built using these models are generally more flexible than those developed with traditional communication models, since the components are loosely coupled and the system can be dynamically reconfigured for changing communication requirements.

4.1 Event Distribution Models

In an event distribution model, a component can generate, or publish, event notifications and can be notified of events that occur. Upon notification, a component may process the information accordingly, perhaps generating another event notification in response. Rather than receiving all published event notifications and discarding unwanted notifications, a component can specify the set of event notifications that it wishes to receive by subscribing for events. Moreover, a component may also remove a subscription for an event when no longer interested in being notified of its occurrence. Components subscribe to event notifications and publish event notifications through an event dispatching service.

In several event distribution models, a component subscribes to event notifications using channel-based or subject-based subscriptions. In channel-based systems, components subscribe and publish to channels. A component can publish event notifications to a particular channel by sending

the notification along a communication path specified by the channel. In subject-based systems, components subscribe to subjects, and components publish event notifications with specific fields indicating the relation to predefined subjects. All event notifications published to the specified channel or subject are delivered to subscribers of that channel or subject, respectively. While these methods of subscription continue to be utilized in several academic and commercial event notification services [19, 3, 20], a current trend is to subscribe for events using a content-based subscription mechanism. Content-based mechanisms allow for more expressive subscriptions because events can be filtered based on any combination of fields and properties. Some content-based event notification services also allow subscribers to specify predicates that a notification must match [21, 22, 23, 24, 25]. For example, a subscription can be made for events in which an event field corresponding to an employee's salary is greater than or equal to $60,000. Such expressiveness in the subscription mechanism can decrease the number of event notifications that must be propagated, which is key to making the system scalable.

Recent work in developing event distribution systems suggest the need to adapt the model for use in mobile environments [26, 27, 21]. However, to support mobility in an event distribution model, one must address issues concerning the configuration of the event dispatching mechanism, location transparency, and disconnection. In many event distribution models, the event dispatching mechanism is centralized. While such an architecture is acceptable in situations where a wired infrastructure is available, it is not adequate to support ad hoc mobility. Event dispatching models for ad hoc mobility suggest that each component should act as a publisher, a subscriber, and an event dispatcher [27, 21]. Location transparency is offered in the model presented in [21] by using a Uniform Resource Identifier (URI) as a reference to the component, keeping the physical location hidden. Finally, in the models presented in [27, 26], the event dispatching service queues event notifications upon disconnection to prevent "dropped" event notifications.

4.2 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 as connections between hosts are established and dropped. The shared, or federated, 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. Like other event distribution models for mobility, we treat each agent as a publisher, subscriber, and event dispatcher. Our event distribution API is shown in Figure 6.

An event notification can be generated by using **Publish(Tuple notification)** where **notification** is a set of fields that defines an event notification. With **Publish(Tuple notification)**, the event notification specified is placed into the event repository, represented by a tuple space. The notification is made available to all agents that are connected to the originator of the notification through transient sharing of local tuple spaces.

An agent must register for an event in order to receive notification of its occurrence. To register for an event notification, an agent uses **Subscribe(ITuple template, LimeEventListener listener)**. The parameter **template** is a set of fields that specifies a pattern for event notifications in which the subscriber is interested. This template, represented as a tuple, 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 by an agent to remove its subscriptions for event notifications. **Unsubscribe(ITuple template, LimeEventListener listener)** is used by an agent to remove a single, specific registered subscription. To ensure that the correct subscription is removed, both the event notification pattern and the handle to code used in the original subscription are provided as parameters. Use of this operation results in the removal of the subscription previously registered by the calling agent upon the event notification pattern **template** with callback function **listener**, meaning that **listener** will no longer be executed as a result of matching a published event notification with **template**. The **UnsubscribeAll(ITuple template)** operation removes *all* subscriptions previously registered upon the event notification pattern provided as a parameter. This means that for every subscription previously made by an agent on the event notification pattern **template**, the callback function specified in each subscription will no longer be executed on that agent when a published event notification matches **template**. In both **Unsubscribe(ITuple template, LimeEventListener listener)** and **UnsubscribeAll(ITuple template)**, the event notification pattern provided as a parameter in the unsubscribe operation must exactly match the event notification pattern used in a previously registered subscription for the unsubscription to be successful. Also, neither operation allows negative unsubscriptions, that is, it is not possible to unsubscribe for a pattern for which no subscription has been registered with the intent of never receiving event notifications that match the pattern. In both operations, if an agent unsubscribes for an event notification template for which it had not previously subscribed, the unsubscription is ignored but the user is notified.

Though location transparency is generally desirable in a mobile environment, there are situations in which it may be beneficial to limit the scope of event notifications in which a component is interested. Therefore, we include a **Subscribe(location source, ITuple template, LimeEventListener listener)** operation which allows an agent to subscribe for event notifications that are generated by a specific agent. The parameter **source** identifies an agent by the specified location and is used to restrict delivery of notifications to those originated by that agent.

4.3 Implementation

Since LIME provides a reactive programming model, the event distribution model has a straightforward implementation in terms of LIME primitives. The functions offered by the event distribution model are little more than wrapper functions for using the **in**, **out**, and reactive operations in LIME. Since LIME provides mechanisms to support physical and logical mobility, no major extensions were necessary to facilitate event distribution for a mobile environment. The implementation is discussed in more detail below.

Publish(Tuple notification)
— generates an event notification defined by **notification**.
The event notification is placed in the event repository.

Subscribe(ITuple template, LimeEventListener listener)
— registers an agent for an event notification appearing in the event repository matching the pattern specified by **template**. Upon matching an event notification, the code specified by **listener** is guaranteed to eventually be executed.

UnsubscribeAll(ITuple template)
— deregisters an agent for event notifications matching **template**.
After unsubscribing in this way, an agent will no longer execute *any* code fragment as a result of the appearance of a matching event notification in the event repository.

Unsubscribe(ITuple template, LimeEventListener listener)
— deregisters an agent for event notifications matching **template**.
After unsubscribing in this way, an agent will no longer execute the code fragment specified by **listener** as a result of the appearance of a matching event notification in the event repository.

Subscribe(location source, ITuple template, LimeEventListener listener)
— registers an agent for an event notification matching that specified by **template**, generated by the agent specified by **source**. Upon receiving a matching event notification, the code specified by **listener** will eventually be executed.

Figure 6: Event Distribution Veneer API

As mentioned before, communication is achieved through the use of transiently shared tuple spaces. In LIME, sharing tuple spaces between agents is possible only when agents have tuple spaces that share the same name. 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 code fragment, i.e., a callback function. The basic idea behind a reaction is that the callback function will be executed upon the appearance of a tuple that matches the specified pattern.

Generally, an agent will subscribe to an event notification that is published in the **EventTupleSpace** of an agent on another host. Since performing the callback function over multiple hosts while maintaining the atomicity requirements of reactions would require a distributed transaction, we use the weak reaction provided in LIME, in which atomicity requirements of reactions are relaxed. A weak reaction guarantees that upon appearance of an event notification in the **EventTupleSpace** matching the provided event notification 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.

Weak reactions interact with the publication of event notifications as follows. When an event notification is placed in the **EventTupleSpace**, the reactions registered by an agent in its subscriptions will fire one by one. A reaction is selected non-deterministically for evaluation, and the event notification pattern given in a subscription is compared with the newly inserted tuple representing the event notification. If the two match, then the code fragment associated with the reaction, **listener**, will eventually execute. Nondeterministic selection and evaluation of registered reactions continues until all registered reactions have been evaluated.

We place restrictions on the firing of weak reactions. Every time an event notification tuple is published that matches a template specified in a subscription, the reaction associated with the subscription should fire as long as the same event notification has not previously triggered the reaction. In LIME, this is accomplished by defining the reaction as occurring once per tuple. When reactions are defined in this way, agents perform bookkeeping operations as reactions are fired, recording the identifier associated with 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 to ensure that the reaction does not fire twice upon tuples having the same identifiers.

It is possible that an agent will receive duplicate event notifications due to overlapping subscriptions. For example, an agent could subscribe for an event notification pattern that specifies an employee event by a formal representing a social security number and a formal representing employment status (e.g., the tuple (formal SSN, formal Status), as well as subscribing for an event notification pattern that specifies an actual value representing the employee's social security number and a formal representing employment status (e.g., the tuple (555-55-5555, formal Status)). These subscriptions are distinct; the former results in receiving information about any employee and the latter results in receiving information about a specific employee. If an event notification such as (555-55-5555, "Newly hired") is placed in the **EventTupleSpace**, the agent in this example would receive two copies of the same event.

As mentioned previously, an agent can require multiple subscriptions for the same event notification pattern to execute different code fragments upon receiving a matching event notification. To register multiple subscriptions on the

same event pattern, the pattern provided as a parameter must exactly match a pattern in an existing subscription. To avoid mistakes in providing exactly the same pattern, the programmer should store an event notification pattern in a variable. The reference will be used as a parameter in all future subscribe operations, rather than registering subscriptions by value.

To aid in the implementation of the **UnsubscribeAll(ITuple template)** operation, the subscribe operation stores the subscriptions in a hash table, using **template** as a key.

We provide the ability to unsubscribe in a manner similar to that used to subscribe for an event notification. **Unsubscribe(ITuple template, LimeEventListener listener)** and **UnsubscribeAll(ITuple template)** are realized by deregistering 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, specifically an event notification pattern and a handle to code to be executed upon the publication of a matching event notification. The hash table in which subscriptions are stored is searched to determine if such a subscription exists. If so, then that subscription is removed using the LIME **removeWeakReaction** operation. In the **UnsubscribeAll(ITuple template)** operation, the programmer specifies the event notification pattern for which it no longer is interested in receiving notifications. A search of the hash table which stores subscriptions is performed 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. Continuing with the earlier example, an agent that maintains both subscriptions, but unsubscribes only for the event notification pattern (555-55-5555, formal Status), would still continue to receive the notification (555-55-5555, "Newly hired"). This is because it still maintains the other subscription that would result in the agent receiving notification concerning any employee's employment status.

In both unsubscribe operations, the provided parameter **template** must exactly match the event notification pattern for which a subscription in the hash table exists. While this approach may seem to be error-prone with respect to submitting the correct event notification pattern as a parameter in unsubscriptions, using a reference to an event notification pattern used in a subscription rather than the actual event notification pattern can reduce the possiblity of mistakes in using unsubscribe operations.

Publish(Tuple notification) uses the LIME **out** primitive to place the event notification in the **EventTupleSpace**. As indicated by the type **Tuple**, each field of **notification** corresponds to an actual (a value). Since we assume that resources are limited in a mobile environment, and, in addition, we wish to provide a scalable implementation, it is important to remove event notifications from the **EventTupleSpace** that are no longer needed. 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 the immediate removal of an event notification after its placement in the **EventTupleSpace** would prevent the delivery of the notifications to subscribers. However, because reactions are used in subscriptions, this is not the case. When an agent generates an event notification, the reactions will fire upon the placement of the event notification tuple in the tuple space, all subscribers will be given the event notification, and then the event notification tuple will be removed from the **EventTupleSpace**. All reactions registered for a given event notification pattern are guaranteed to occur if an event notification matching that pattern appears in the tuple space.

4.4 Issues

One issue of interest is that of 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. Since all event notifications published to the federated tuple space are immediately removed, and subscriptions are implemented using reactions, the agent will not receive event notifications that were generated before or during the time that the agent was disconnected.

While supported in other models, 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 those agents that are connected. Upon disconnection due to physical movement of hosts, reactions are implicitly deregistered by the LIME system and are implicitly re-registered when a communication link is established once more. Likewise, in the event distribution veneer, an agent implicitly unsubscribes for all event notifications upon disconnection, and implicitly subscribes again upon reconnection. Second, it is a reasonable assumption that, in some cases, event notifications should not be received while a host or agent is disconnected. Consider, for instance, a system that uses time-sensitive information. Upon reconnection, the information will likely no longer be useful to the agent. Thus, the events should not be queued for the agent. This is the assumption that we currently make in our event distribution model.

Another point of interest is the subscription mechanism. Our event distribution model provides a form of content based subscription mechanism in that several fields and their contents can be used to filter notifications for "delivery" to the appropriate agents, rather than limiting the filtering process to one field as in subject-based subscription. However, our model does not currently support matching based on predicate evaluation. While the LIME pattern matching mechanism has recently been extended to support the evaluation of predicates in tuples, this method of pattern matching has not yet been incorporated into use with the event veneer.

4.5 Application example

An example application that could be built using the event veneer is a cruise ship activity update program. Consider a cruise ship that keeps its passengers informed about various activities throughout the day. In this application, passengers and staff on the cruise ship are equipped with PDAs, which are loaded with ship activity software that utilizes

the event veneer on top of LIME. Every person on the ship with a PDA loaded with the software will have access to the ship's activity event repository when in connection range. Once all passengers have boarded, the cruise activity director publishes to the event repository a list of activities that will be occurring during the week on the ship. Cruise ship activity events are of a standard form, and are tagged with the cruise ship activity director's name and activity type. Initially, passengers are subscribed to all updates from the cruise activity director. After receiving the list, passengers can select activities from the list in which they are interested. Passengers also have the option to not receive any notifications about activities at all. When submitting their activity update preferences, the passengers will be unsubscribed from all activity notifications identified by the activity director's name and are subscribed for future activity updates according to the activities selected from the list. Later in the week, the activity director will publish updates about the activities to the ship's activity repository. Passengers on the cruise ship will receive updates concerning activities in which they expressed interest.

5. SECURE TRANSPARENT DATA SHARING

Mobile agent systems bring radical changes in application design. While mobile agents offer a high degree of flexibility, their utilization in an application introduces new challenges for the developer. 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 in the system, 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 any kind of protection, applications developed on top of a tuple space infrastructure remain of purely academic interest, since they are easy to tamper with. In such open environments as discussed above, security becomes of heightened importance. Several systems attempted to add security to tuple space coordination of mobile agents. KLAIM (a Kernel Language for Agents Interaction and Mobility) [28] addresses the protection of data through the use of a type system and SECOS [29] adds fine-grained access control to Linda. This latter approach has difficulty scaling from a single machine to a distributed setting due to issues relating to key secrecy. Several systems address the problem of protecting hosts from malicious agents. The D'Agents [30] system uses public-key cryptography to authenticate incoming agents and thus increase the security of hosts. The more difficult problem of protecting agents from malicious hosts led to agents computing with encrypted functions [31, 32]. This presents a model that enables mobile agents to decrypt code and data only if certain conditions in the environment are met or at a specific moment. Administrative domains [33, 34] restrict the execution environment by logically dividing it into nested levels. The scope of a user's operations can be limited to his/her domain. Another approach is to offer more than a single tuple space giving each application the chance to use a separate tuple space. Many models offer this feature but with only limited increases in security.

5.2 A Secure Tuple Space Communication 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 secure and unsecure tuple spaces. The tuple spaces are shared based on their name in the internal representation. This internal representation is essentially the same as the name provided by the programmer if the tuple space is not intended to be secure, or the result of encrypting the name with the password provided by the user, i.e., tuple spaces having the same name but protected by different passwords are not shared.

LIME has a special tuple space called the `LimeSystemTupleSpace`, which contains, among other things, the names of all tuple spaces. Any agent can read from `LimeSystemTupleSpace`, 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. We assume that an agent that has the password is entitled to have full access to the entire tuple space. The model uses symmetric encryption. Whenever a new tuple space is created, the programmer can provide a password to protect it. (Tuple spaces without passwords are still accepted for backward compatibility with older versions of LIME. As in LIME, all agents are allowed full access to unprotected tuple spaces.) 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. This makes it impossible for an agent to use a handle obtained from another agent (even if the latter obtained the handle correctly, using the public name of a tuple space and the correct password).

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. However, secure tuple spaces are protected from calls originating on hosts that use older (insecure) versions. Older APIs will send requests unencrypted and will not be served if the tuple spaces they refer to are password protected.

SecureLimeTupleSpace(java.lang.String name, java.lang.String password)
— creates a new secure tuple space using the public tuple space name and the password.
This call places an entry in the **SecurityTable** mapping the mangled name to the password.

Figure 7: The Call that Creates a Secure Tuple Space

The mangled 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 mangled 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, obtaining 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. While the constructors still exist in their previous forms (see LimeTupleSpace class), a new one was added, to take an extra parameter: the password (Figure 7). As mentioned above once the agent has the handle to the tuple space, it is free to access it unrestrictedly. Since this handles are 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.

A secure tuple space is created with a name that results from a encrypting the provided (clear) name with the password. The encrypted name appears in the `LimeSystemTupleSpace`. 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 unsecure or "S" if it is a secure tuple space. As discussed previously, our implementation ensures that this name cannot be used incorrectly. A secure tuple space name can only be used if generated correctly. Having this extra step before actually creating and using the tuple space name ensures that names copied from `LimeSystemTupleSpace` cannot be used directly.

The LIME server was augmented with a **SecurityTable** that stores entries of the form [encrypted name, password]. An entry is added to this table when a new tuple space is created. 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 [35]. When an agent executes a call, an interceptor catches it and performs the encryption/decryption before forwarding it to the network or LIME system, respectively. Applying the interceptor pattern, all calls (Linda-like or reactions) are treated similarly, even though they have different implementations. The encryption is realized using the DES encryption algorithm.

5.3.1 Local Execution

Local operations execute only on the projection of the tuple space local to the host where the agent runs. Once the call is identified as being local to the issuer it is directly executed, with no further verifications.

5.3.2 Remote Execution

Remote operations are executed on some specified projection of the federated tuple space that includes hosts besides the one on which the requesting agent runs. There are two different types of messages that can carry the request to a remote host: messages that carry reactions (called **uponmessages**) and messages that carry simple tuple space operations (called **regularmessages**). When an agent executes a call, an interceptor catches it, analyzes the tuple space that the message refers to and, if its name is found in the **SecurityTable**, it will encrypt the message using the password the agent provided when the tuple space was created. The system then forms a packet consisting of the encrypted operation and the encrypted name of the tuple space the operation is being performed on and forwards this packet to the other involved hosts. At the receiving end, a second interceptor catches the incoming packet, looks up the name in the local **SecurityTable**, and uses the corresponding password from the table to decrypt the message, if such a password exists. If the decryption is successful, the operation is performed on the local host. If it is unsuccessful, an exception is generated. The return of results from tuple space operations is handled in the same manner. Figure 8 shows how the interceptors function to secure tuple space communication.

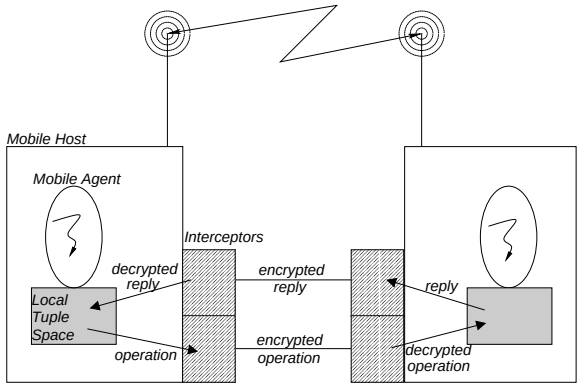

Figure 8: Interceptors catch messages and encrypt them before sending and decrypt after receiving.

5.4 Issues

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

5.5 Application Example

To illustrate one representative use of secure tuple space sharing, let's consider a tollbooth application. A car equipped with wireless communication capabilities approaches a tollbooth on a highway. The payment should be done automatically, as the car passes by the tollbooth. The car will transmit the driver's credit card information to the tollbooth. The tollbooth is connected to a network and can verify the credit card information as well as obtain other information about the driver, like a password that will be used for secure communication (we make here the reasonable assumption that the driver has previously registered for this type of payment and made all the arrangements with the company operating the tollbooth, including setting up a password). As the car approaches the tollbooth, the car and the tollbooth will begin a dialog using a public, predefined tuple space. This will help set up the private communication channel which will be used for the payment. The password that the driver provided when she/he registered for this service is used both to verify driver's identity and to encrypt the communication. The secure tuple space is created by both parties and shared if all security conditions are met (i.e., both parties used the same parameters to generate the encrypted name of the tuple space).

6. CONCLUSIONS

The paper describes three coordination veneers designed to support the development of applications over ad hoc networks. The veneers that we have deveoped not only can be used to develop applications that require specific types of coordination, but can also be combined to offer support for even more complex applications. When combining the security veneer with another veneer, however, it is required that the tuple spaces are created using `SecureLimeTupleSpace` and not `LimeTupleSpace`.

Interaction between the service provision veneer and the event notification veneer combined with the implementation of the safe distance concept leads to improved performance of the applications involving services. If an application uses a service advertised by an agent on some host in the group, there's no control on the behavior of the proxy and the server that implement the service in case of disconnection. The proxy can fully implement the service. In this case the communication back to its server is unimportant. The communication between the two can be done via LIME tuple spaces. In this case, if the disconnection occurs either or both parties (proxy and server) could block indefinitely, waiting on a blocking call that cannot complete until the two hosts engage again. This doesn't affect the system but the application freezes for an undetermined period of time. Another possibility is that the proxy and the server communicate using another protocol that depends on a reliable network, like Java RMI. In this case as well as any other case where the two parties use their own private protocol, the disconnection can the detected only when it actually occurs and there's nothing much to be done beside an effort to recover from a socket exception. By combining the event notification veneer with the service provision model and using them on top of LIME, these situations can be avoided. A special system event can be defined to be generated when a host crosses the safe distance and is about to be disconnected. Applications that register their interest in this event are able to perform some operations that will prevent an inconsistent state or a breakdown when the actual disconnection occurs. If the implementation of a service takes advantage of this system event, the proxy and the server can "freeze" their cooperation by saving the state of their work and safely closing down their communication channels, as well as alert the user about the reason for stopping computational activity.

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 in the ad hoc network. Equally important is the fact that all these veneers were constructed with minimal effort over an existing coordination middleware (LIME). On one hand, we have been able to demonstrate the feasibility of employing specialized coordination middleware in software development while, on the other hand, this work offers additional evidence regarding the expressive power of LIME and its underlying model.

All three veneers have been fully implemented and will soon be publicly available. In the future, we plan to develop applications that utilize the veneers independently, as well as the tollbooth application that utilizes all three veneers. We plan to use these applications to evaluate the usefulness, usability, and performance of each veneer and their combination.

Acknowledgements

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

7. REFERENCES

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

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

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

[4] IBM. T Spaces. http://www.almaden.ibm.com/cs/TSpaces/, 2002.

[5] A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[6] Roman G.-C., Huang Q., and Hazemi A. Consistent group membership in ad hoc networks. In *Proceedings of the 23rd International Conference in Software Engineering (ISCE)*, 2001.

[7] J. Newmarch. *Guide to Jini Technologies*.

http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml, 2001.

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

[9] E. Guttman, C. Perkins, Sun Microsystems, J. Veizades, M. Day, and Vinca Corporation. Internet Working Group RFC 2608: Service location protocol, version 2. http://rfc.sunsite.dk/rfc/rfc2608.html, 1999.

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

[11] 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, 2001.

[12] T. Berners-Lee, MIT/LCS, R. Fielding, U.C. Irvine, L. Masinter, and Xerox Corporation. Uniform Resource Identifiers (URI): Generic syntax. http://rfc.sunsite.dk/rfc/rfc2396.html, August 1998.

[13] W3Schools. Welcome to XML school. http://www.w3schools.com/xml/default.asp, 2001.

[14] R. Droms. Resources for DHCP. http://www.dhcp.org/, January 2002.

[15] B. Pascoe. Salutation architectures and the newly defined service discovery protocols from Microsoft and Sun. http://www.salutation.org/whitepaper/Jini-UPnP.PDF, 1999.

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

[17] Sun Microsystems. Java Remote Method Invocation. http://java.sun.com/products/jdk/rmi/, 2002.

[18] G.P. Picco. μCode: A lightweight and flexible mobile code toolkit. In *Proceedings of the 2^{nd} International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes on Computer Science*, pages 160–171. Springer-Verlag, September 1998.

[19] H. Holbrook and D. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *SIGCOMM*, pages 65–78. ACM, September 1999.

[20] Talarian Corporation. Rapid infrastructure development for real-time, event-driven applications, April 1998.

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

[22] Object Management Group. CORBA*services*: Notification service specification. http://www.omg.org/technology/documents/formal/notification_service.htm, June 2000.

[23] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19^{th} International Conference on Distributed Computing Systems*, pages 262–272, 1999.

[24] M. Wray and R. Hawkes. Distributed virtual environments and VRML: an event-based architecture. In *Proceedings of the 7^{th} International WWW Conference*, 1998.

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

[26] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proceedings of the 2^{nd} International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, April 2001.

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

[28] 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.

[29] J. Vitek, C. Bryce, and M. Oriol. Coordinating agents with secure spaces. In *Proceedings of Coordination '99*, Lecture Notes on Computer Science. Springer Verlag, May 1999.

[30] 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 *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[31] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agent Security*, Lecture Notes in Computer Science, pages 44–60. Springer-Verlag, 1998.

[32] J. Riordan and B.Schneier. Environmental key generation towards clueless agents. In *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 15–24. Springer-Verlag, 1998.

[33] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378, pages 140–155. Springer-Verlag, 1998.

[34] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *ICCL Workshop: Internet Programming Languages*, pages 47–77, 1998.

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