Report Number: WUCSE-2003-12

2003-03-17

A Lightweight Coordination Model and Middleware for Mobile Computing

Chien-Liang Fok and Gruia-Catalin Roman

Limone is a new coordination model and middleware that enables rapid application development for wireless ad hoc networks entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, Limone performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies that are customizable by the application. This asymmetry among participants represents a new direction in coordination research and is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. The coordination context is... Read complete abstract on page 2.

A Lightweight Coordination Model and Middleware for Mobile Computing

Chien-Liang Fok and Gruia-Catalin Roman

Complete Abstract:

Limone is a new coordination model and middleware that enables rapid application development for wireless ad hoc networks entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, Limone performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies that are customizable by the application. This asymmetry among participants represents a new direction in coordination research and is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. The coordination context is limited to the specific needs of the individual agent and its coordination activities are restricted to tuple spaces owned by peers present in the acquaintance list. Designed for wireless ad hoc networks, Limone tailors Linda-like primitives to address the challenges of mobile environments. This entails the elimination of remote blocking operations and the addition of timeouts to all distributed operations since disconnection with the affected agents may occur at any time. It also entails the addition of reactions that are triggered by the presence of information of interest on agents listed in the acquaintance list. Finally, to ensure performance and ease of deployment on small devices the granularity of atomic operations and the assumptions about the environment have been minimized. This paper introduces Limone, explains its key features, illustrates its usage, and explores its effectiveness as a software engineering tool.

2003-12

A Lightweight Coordination Model and Middleware for Mobile Computing

Authors: Fok, Chien-Liang; Roman, Gruia-Catalin

March 17, 2003

Abstract: Limone is a new coordination model and middleware that enables rapid application development for wireless ad hoc networks entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, Limone performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies that are customizable by the application. This asymmetry among participants represents a new direction in coordination research and is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. The coordination context is limited to the specific needs of the individual agent and its coordination activities are restricted to tuple spaces owned by peers present in the acquaintance list. Designed for wireless ad hoc networks, Limone tailors Linda-like primitives to address the challenges of mobile environments. This entails the elimination of remote blocking

A Lightweight Coordination Model and Middleware for Mobile Computing

Chien-Liang Fok and Gruia-Catalin Roman
Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130-4899, USA
{liang, roman}@cse.wustl.edu

ABSTRACT

Limone is a new coordination model and middleware that enables rapid application development for wireless ad hoc networks entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, Limone performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies that are customizable by the application. This asymmetry among participants represents a new direction in coordination research and is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. The coordination context is limited to the specific needs of the individual agent and its coordination activities are restricted to tuple spaces owned by peers present in the acquaintance list. Designed for wireless ad hoc networks, Limone tailors Linda-like primitives to address the challenges of mobile environments. This entails the elimination of remote blocking operations and the addition of timeouts to all distributed operations since disconnection with the affected agents may occur at any time. It also entails the addition of reactions that are triggered by the presence of information of interest on agents listed in the acquaintance list. Finally, to ensure performance and ease of deployment on small devices the granularity of atomic operations and the assumptions about the environment have been minimized. This paper introduces Limone, explains its key features, illustrates its usage, and explores its effectiveness as a software engineering tool.

1. INTRODUCTION

Mobile computing devices having wireless capabilities have experienced rapid growth in recent years due to advances in technology and social pressures from a highly dynamic society. Many of these devices are beginning to allow for the formation of ad hoc networks in which connected communities are formed without the aid of a wired network infrastructure. Applications for ad hoc networks are expected to quickly grow in importance because they address challenges set forth by several important application domains. By eliminating the reliance on the wired infrastructure, ad hoc networks can be rapidly deployed in disaster situations where the infrastructure has been destroyed or in military applications where the infrastructure belongs to the enemy. Ad hoc networks are also convenient in day-to-day scenarios where the duration of the activity is too brisk and localized to warrant the establishment of a permanent infrastructure.

The salient properties of ad hoc networks create significant new challenges for the application developer. The inherent unreliability of wireless signals and the mobility of nodes result in frequent unannounced disconnections and message loss. In addition, mobile devices are typically limited in battery and computational power, further exacerbating the difficulties associated with application development. The limited functionality of mobile devices often leads to strong mutual dependencies. Devices may have to cooperate to achieve a common goal, resulting in a greater need for coordination support. For example, in a planetary exploration setting ad hoc networking enables miniature rovers each equipped with a single sensor to perform experiments that demand data from any arbitrary combination of sensors.

Mechanisms that address the complexities of ad hoc networks include enhancements to the operating system, specialized languages, and middleware. Among these, middleware is the most popular. Operating systems are tightly integrated with low-level communication services (e.g., TCP sockets) and expose too many details in the design of distributed applications. The development and use of new programming languages typically require too great an investment and entail too high a risk. In contrast, middleware provides higher level abstractions while minimizing risk by using the existing software infrastructure. When designed properly, middleware can divert attention from mundane concerns like protocol development to more fruitful areas like models, algorithms, and applications.

Of the numerous models for mobile environments that have been developed, LIME [1, 2] is the only model to support ad hoc mobility. It is a coordination model that uses distributed transactions to provide strong atomicity and consistency guarantees provided that certain assumptions about the environment are met. Unfortunately, many of these assumptions, such as the ability to predict when a disconnection is about to occur, are impossible to meet. Furthermore,

it does not make sense for applications that rely on strong atomicity and consistency guarantees (e.g., anything dealing with monetary transactions) to use wireless ad hoc networks. Thus, applications for such an environment should not require the high levels of atomicity and consistency guarantees that LIME provides. It is this observation that motivated this research effort.

In this paper we introduce Limone (Lightly-coordinated Mobile Network), a new lightweight coordination model and middleware for mobile environments supporting logical mobility of agents and physical mobility of hosts. Limone agents are software processes that represent the unit of modularity, execution, and mobility. In a significant departure from existing coordination research, the individuality of each agent is emphasized by focusing on asymmetric interactions among agents. Each agent contains a new abstraction called the *acquaintance list* which defines a personalized view of remote agents within range. For each agent, Limone performs remote agent discovery and updates its acquaintance list according to policies that are customizable by the application.

As in most coordination models, traditional Linda-like primitives over tuple spaces [3] facilitate the coordination of agent activities. However, Limone allows each agent to maintain strict control over its local data, provides advanced pattern matching capabilities, permits agents to restrict the scope of their operations, and offers a powerful repertoire of reactive programming constructs. The autonomy of each agent is maintained by the exclusion of transactions and remote blocking operations. Furthermore, Limone ensures that all distributed operations contain built-in mechanisms to prevent deadlock in the case of packet loss or disconnection. By emphasizing minimality of concepts and feasibility, Limone is resilient to message loss and unexpected disconnection. This allows Limone to function in realistic ad hoc environments where existing models cannot.

The paper starts with an overview of Limone in Section 2. Section 3 presents a motivational example that describes how Limone can be used to provide a spatially-directed multicast. Section 4 presents Limone's run-time environment which describes the functionality of the constructs Limone provides for the application. We then proceed with a discussion in Section 5 of the assumptions and vulnerabilities of Limone, and elaborate on its implementation. We end with a section on related work (Section 6) that describes how Limone can be used to implement existing coordination models, and we draw conclusions in Section 7.

2. MODEL OVERVIEW

The coordination model we present in this section has been shaped by a set of highly pragmatics software engineering concerns. Foremost among them is the desire to facilitate rapid development of mobile applications in a mobile setting under realistic assumptions regarding the environment. While other models are willing to rely on strong assumptions, such as precise knowledge about the status of communication links, we readily acknowledge the unpredictable and dynamic nature of wireless ad hoc networks. As such, we do not presume to know when communication links go up or down or the range of the wireless signals. The model starts with the premise that a single round trip message exchange is reliable and, under this minimalist assumption, it offers a precise and reasonable set of functional

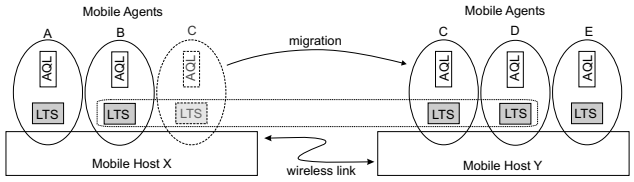

Figure 1: An overview of the Limone model. Agents are represented as ovals. Each agent owns a local tuple space (LTS) and an acquaintance list (AQL). In this example, agent C is shown as migrating to host Y without a change in its acquaintance list, which consists of B and D. The dotted rectangle surrounding the tuples spaces of agents B, C, and D highlight the tuples spaces that are accessible from C.

guarantees. Moreover, even if this assumption is not met, the system continues to function and tolerates possible message losses through the use of timers.

The willingness to accommodate a high degree of uncertainty about the physical state of the system raised important research questions regarding the choice of coordination style and associated constructs. A minimalist philosophy, combined with the goal of achieving good levels of performance, led to the emergence of a novel model whose elements appear to support fundamental coordination concerns. Central to the model is the organization of all coordination activities around an acquaintance list that reflects the current local view of the global operating context and whose composition is subject to dynamically changing admission policies. From the application point of view all interactions with other components take place by referring to individual members of the acquaintance list. All operations are content-based, but can be active or reactive. This perspective on coordination, unique to Limone, offers an expressive model that enjoys an effective implementation likely to transfer to small devices.

Limone assumes a computational model consisting of mobile devices (hosts) that form ad hoc networks; mobile agents that reside on hosts but can migrate from one host to another; and data owned by agents that is shared through Linda-like tuple spaces [3]. The relationship between hosts, agents, and tuple spaces is shown in Figure 1. The features of Limone can be broadly divided into four general categories: context management, explicit data access, reactive programming and code mobility.

Central to the notion of context management is an agent's ability to discover neighbors and to selectively decide on their relevance. Limone provides a beacon-based discovery protocol that informs each agent of the arrival and departure of other agents. Limone notifies each agent of its relevant neighbors by storing them in individualized acquaintance lists, where relevance is determined using an *engagement policy* that is customizable by the application. Since each agent has different neighbors and engagement policies, the context of each agent may differ from that of its peers.

Existing coordination models for mobility in ad hoc environments presume a symmetric and transitive coordination relation among agents that is not scalable. If every node must coordinate with every other node, the total computation required is the square of the number of nodes. Furthermore, as the number of nodes increases, the likelihood

that some nodes move out of range also increases, generating frequent configuration changes. By allowing an agent to restrict coordination to agents it is interested in, Limone is better able to scale to dense ad hoc networks as well as to devices with limited memory resources. For example, if an agent is surrounded by hundreds of other agents but is interested only in two of them, it can concentrate on these two and ignore the rest, minimizing wasted memory and processor cycles. In addition, this asymmetry increases the level of decoupling among agents and results in a more robust coordination model that requires fewer assumptions about the underlying transport layer [4].

Limone accomplishes explicit data access in a manner similar to that employed by most other coordination models. Each agent owns a single tuple space which offers operations for inserting and retrieving tuples through a pattern-matching mechanism. Explicit data access spans at most two agents. The agent initiating the data access (called the reference agent) must have the remote agent in its acquaintance list. Our initial approach was to allow the reference agent to perform operations on remote agent's tuple space. But upon further review, we decided for security and simplicity reasons that the reference agent can only *request* a remote agent to perform the operation for it. By doing this, each agent maintains full control over its local data and can implement policies for rejecting and accepting requests from remote agents. This is accomplished using a *remote operation manager*.

The remote operation manager controls which requests from remote agents are actually performed. Implementation-wise, this manager is an interface that the application can implement and configure Limone to use. By default, all requests are allowed to be performed. This manager greatly enhances the expressiveness of Limone since it can be customized to perform relatively complex tasks such as those dealing with security.

Suppose each agent creates a public/private key pair and publishes its public key in a "read-only" tuple. The read-only nature of this tuple can be enforced by the remote operation manager, which can use the public and private keys for secrecy and authentication. Suppose a reference agent wishes to place a tuple onto a remote agent's tuple space. To do this, it can encrypt the data first by its private key, then by the remote agent's public key. The remote agent can conclude with some degree of confidence that the tuple is secret if it is able to decrypt it using its private key. It can also conclude with some degree of confidence that the tuple was sent by the reference agent if it can decrypt it using the reference agent's public key. While the analysis of the degree of security is outside of the scope of this paper, the intuition behind why total confidence is not possible is due to the possibility of the man-in-the-middle attack where the attacker injects fake public keys, confusing either or both of the agents. The point is simply that the remote operation manager can be configured by the application to perform complex tasks, e.g., authentication, to decide whether a particular operation should be performed.

Limone uses a single tuple space within each agent because it is not limiting. Limone can mimic the behavior of multiple tuple spaces á la LIME by utilizing special fields within each tuple. This can be done without hurting time complexity since the tuple space may be implemented as a hash table keyed by a field in the tuple. Limone can also mimic a single shared tuple space per host á la MARS by setting the engagement policy to consider only agents on the local host.

Reactive programming constructs enable an agent to automatically respond to the appearance of particular tuples within the tuple spaces of agents in its acquaintance list. Two state variables within each agent, the *reaction registry* and *reaction list,* support this behavior. A reference agent registers a reaction by placing the reaction into its reaction registry. Once registered, Limone automatically propagates the reaction to all agents in the acquaintance list that satisfy certain properties specified by the reaction (e.g., agent location). At the receiving end, the operation manager is used to determine whether to accept the reaction. If accepted, the reaction is placed into the reaction list, which contains the reactions that apply to the local tuple space.

When the tuple space contains a tuple satisfing the trigger for a reaction in the reaction list, the agent that registered the reaction is sent a notification consisting of a copy of the tuple and an value identifying which reaction was fired. If this agent receives this notification, it executes the code associated with the reaction atomically. This mechanism, originally introduced in Mobile UNITY [5], is distinct from that employed in traditional publish/subscribe systems in that it reacts to state properties rather than to data operations. For instance, when a new agent is added to the acquaintance list, its tuples may trigger reactions regardless of whether the new agent performed any operations.

Code mobility is supported in Limone by allowing agents to migrate from one host to another. When an agent migrates, Limone automatically updates its context and reactions. There are many benefits to allowing an agent to migrate. For example, if a particular host has a large amount of data, an agent that needs to operate on it over an extended period of time can relocate to the host holding the data and thus have reliable and efficient access to it despite frequent disconnection among hosts. Another example of agent mobility is software update deployment. Suppose an agent is performing a certain task and a developer creates a new agent that can perform the task more efficiently. The old agent can be designed to shutdown when the new agent arrives. Thus, simply having the new agent migrate to the same host as the old agent updates the application. To date, such updates are common practice on the Internet. However, agent migration promises to be even more beneficial in the mobile setting.

3. MOTIVATING EXAMPLE

In this section we illustrate some of the capabilities of Limone by focusing on a simple problem involving a geocast [6]. Consider a source agent and a group of agents as shown in Figure 2. Each dot is an agent on a separate host physically distributed in space as shown in the figure. The lines connecting two agents indicate the existence of a communication link between them. In this example, the distinctively marked agent in the lower-left corner needs to multicast a message to all agents located in the rectangle appearing in the upper-right corner of the figure. The dotted arrows indicate the path the message takes in reaching the destination agents.

Geocast can be easily implemented in Limone using a combination of reactions and explicit data accesses. Suppose the initiating agent places a message in the form of a tuple containing a destination location and data into its

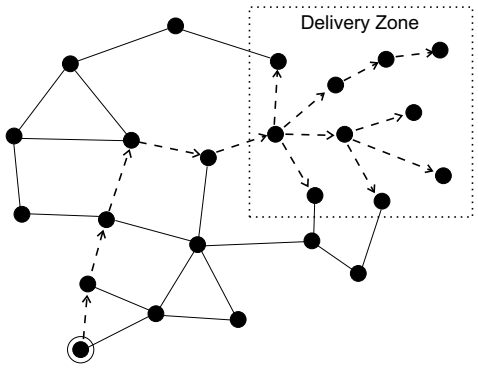

Figure 2: This figure shows the topology of an ad hoc network and an example geocast application. For illustration purposes, the topology is fixed. Each dot is an agent. In this example, the circled agent sends a message to all agents encompassed within the rectangle. The arrows depict the path by which agents along the way pull the message until it reaches the delivery zone, at which point the message is propagated to all nodes in the zone using a reaction mechanism.

tuple space. Special "delivery" agents have reactions sensitive to this tuple. As the delivery agents move, they engage with neighboring agents. If the neighboring agent has a message tuple, the delivery agent's reaction will fire. When this occurs the delivery agent will consider its present location, and the message's present and destination locations. If the delivery agent is closer to the destination, it will pull the tuple from its current location and place it into its own tuple space.

In Figure 2, the dotted arrows depict the path of the message from the origin to the destination agents under a simplified scenario in which no movement occurs. Assuming agents move randomly and eventually encounter other agents, the message will gradually move closer to its destination and eventually reach it. While the message is in transit, multiple remote agents may react to the tuple at each step. This is not a problem because tuple removal is done atomically by the agent that holds the tuple, meaning only one agent will successfully grab the tuple. When the message reaches one of the destination agents, it can place the tuple into its own tuple space firing the reactions of other destination agents. These destination agents can repeat the process causing more destination agents to react to the message. Eventually, all of the destination agents will receive a copy of the message.

The success of this implementation depends upon essential features of the mobile system, including movement patterns and the probability of certain encounters among agents. It is possible for a message to be grabbed by a delivery agent, only to move away from the destination. However, the application will continuously try to move the message towards the destination and should eventually succeed provided there are enough delivery agents moving towards the destination. *The entire geocast implementation entails only one replicated delivery agent whose code consists of essentially one reaction, one input operation, and one output operation.* The reaction is used to notify the agent of a message tuple; the input

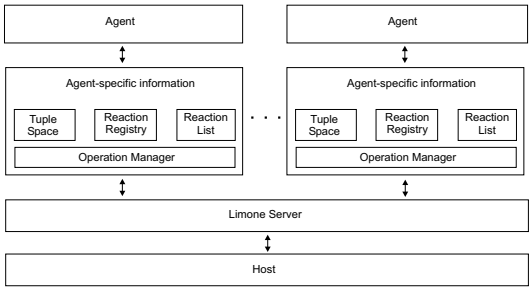

Figure 3: The overall structure of Limone.

operation is used to pull the message towards the correct region in space; and the output operation is used to further transmit or propagate the message. An example implementation of geocast is given in Section 4. An analysis of its performance is extremely complex and is outside the scope of this paper. The point is: from the application developer's perspective, implementing geocast in a wireless ad hoc environment using Limone is trivial.

4. RUN-TIME ENVIRONMENT

Limone provides an environment for agents to operate via the Limone Server, a software layer between the agent and the underlying network transport layer. By using different ports, multiple Limone servers may operate on a single host. However, for the sake of simplicity, we will talk as if each host were restricted to have a single Limone server.

An application uses Limone by interacting with an agent. Each agent contains a tuple space, acquaintance list, reaction registry, reaction list, and operation manager. The overall structure of Limone is shown in Figure 3. An agent allows the application to customize its profile, engagement policy, and operation manager. An agent's profile is a set of objects that describe its properties. Its engagement policy specifies which agents are of interest based on their profiles. The operation manager specified which remote operation requests are allowed to be executed. This section describes how Limone fulfills its responsibilities and is organized around the key elements of the run-time environment, i.e., agent discovery, tuple space management, reactions, and agent mobility.

Discovery Mechanism. Since network connectivity between hosts in ad hoc networks can form and break at any time, Limone provides a *discovery protocol* based on beacons to allow an agent to discover the arrival and departure of other agents.

The beaconing mechanism is the most costly but necessary construct in Limone because it requires periodic broadcasts, consuming a significant amount of network bandwidth, processor resources, and battery power. Each beacon contains a *profile* for each agent running on top of the particular Limone server. A profile is a collection of triples each consisting of a property name, type, and value. The two system-defined entries include the host the agent is on and a unique agent identifier. Additional entries can be added by the application. When the Limone server receives a beacon, it forwards it to each of its agents.

ABSTRACT STATE:

— A set of profiles, $\{p_1, p_2, \ldots\}$

INTERFACE SPECIFICATION:

boolean add(Profile profile)
— Adds an agent's profile into the list.

void clear()
— Removes all profiles from the acquaintance list.

boolean contains(AgentID aID)
— Returns true if the list contains a profile that has the specified AgentID.

Profile[] getApplicableAgents(ProfileSelector[] pss)
— Returns all of the profiles within the list that match any of the specified profile selectors.

void remove(ProfileSelector ps)
— Removes the matching profile(s) from the list.

Figure 4: Acquaintance list.

When an agent receives a beacon, it takes the profiles within it and passes them to its *acquaintance handler* . The acquaintance handler uses the agent's *engagement policy* to decide which profiles are of interest, and places them in the agent's acquaintance list. Also, if any of the profiles represent agents already in the acquaintance list, the acquaintance handler ensures that the profile in the acquaintance list is up to date. If a profile is not, the acquaintance handler updates the profile and ensures that the new profile satisfies the engagement policy.

Once an agent's profile is added to the acquaintance list, the acquaintance handler continuously monitors the agent's beacons. If a beacon is not received for an application-customizable period of time, the acquaintance handler removes the profile from the acquaintance list.

Another way a remote agent's profile can be removed from the acquaintance list is if a remote operation involving the remote agent fails due to a time-out. If a timeout occurs, it is assumed that the agent is no longer within range, justifying the removal of the agent's profile from the acquaintance list.

The acquaintance list, shown in Figure 4, contains a set of agent profiles representing the agents within range that have satisfied the engagement policy. The addition of a profile into the acquaintance list signifies an *engagement* between the reference agent and the agent represented by the profile. Once the reference agent has engaged with another agent, it gradually propagates its relevant reactive patterns (the non-callback function portion of the reaction) to the remote agent. While the addition of the profile to the acquaintance list is done atomically, the propagation of reactive patterns is gradual. The removal of a remote agent's profile from the acquaintance list signifies *disengagement* between the reference agent and the remote agent. When this occurs, the reference agent removes all the remote agent's reactive patterns from its reaction list. The removal of the profile from the acquaintance list and the reactive patterns from the reaction list is performed atomically, which is possible because it is done locally.

Tuple Space Management. Any data available for coordination among agents is stored in individually owned tuple spaces. Each tuple space contains a set of tuples. Limone tuples contain data fields distinguished by name and store user-defined objects and their types. The ordered list of fields characterizing tuples in Linda is replaced in Limone by unordered collections of named fields. This results in a more expressive pattern matching mechanism that can handle situations in which a tuple's arity is not known in advance. In open systems, this is a highly desirable feature. For example, the following tuple may represent the message in the geocast example:

$$\begin{aligned} tuple\{&\langle \text{“}type\text{”},\ String,\ \text{“}Directed\ Multicast\text{”}\rangle,\\ &\langle \text{“}message\text{”},\ String,\ \text{“}TAKE\ COVER!\text{”}\rangle,\\ &\langle \text{“}destination\text{”},\ GPSCoord,\ (90.45\text{N},\ 34.23\text{W})\rangle,\\ &\langle \text{“}area\text{”},\ Meters,\ 10.5\rangle,\\ &\langle \text{“}deadline\text{”},\ Time,\ 14{:}15{:}30\text{Z}\rangle\} \end{aligned}$$

Agents use templates to specify tuples of interest in the tuple space. A template is a collection of named constraints, each defining a name and a predicate over the field type and value called the *constraint function*. A template matches a tuple if each constraint within the template has a matching field in the tuple, i.e., a field having the same name is present in the tuple and the value and type stored in the field satisfy the constraint function. For example, the following template matches the message tuple give above:

$$\begin{aligned} template\{&\langle \text{“}type\text{”},\ String,\ \text{valEql}(\text{“}Directed\ Multicast''\rangle)\rangle,\\ &\langle \text{“}message\text{”},\ String,\ \texttt{defaultConst}(true)\rangle\}\\ &\langle \text{“}destination\text{”},\ GPSCoord,\ \texttt{defaultConst}(true)\rangle\}^{1}\\ &\langle \text{“}area\text{”},\ Meters,\ \texttt{defaultConst}(true)\rangle\} \end{aligned}$$

The bottom three constraints having default constraint functions that always return *true* specify that all matching tuples must contain fields with the individual names and types (i.e., a field named "*message*" with a value of type *String*, a field named "*destination*" with a value of type $GPSCoord$, and a field named "area" with a value of type $Meters$ must be part of the tuple). Since this template did not contain a constraint named "*deadline*," a tuple need not have this field to match the template. Notice that the tuple may contain more fields than the template has constraints. As long as each constraint in the template is satisfied by a field in the tuple, the tuple matches the template. This powerful style of pattern matching provides a higher degree of decoupling since it does not require prior knowledge of the ordering of fields within a tuple nor its arity to create a template for it.

Local Tuple Space Operations. The operations allowed on the local tuple space are shown in Figure 5. The **out** operation places a tuple into the tuple space. The operations **in** and **rd** block until a tuple matching the template appears in the tuple space. When this occurs, **in** removes and returns the tuple, while **rd** returns a copy without removing it. The operations **inp** and **rdp** are the same as **in** and **rd** except they do not block. If no matching tuple exists within the tuple space, ε is returned. The operations **ing** and **rdg** are similar to **in** and **rd** except they find and return *all* matching tuples within the tuple space. Similarly, **ingp** and **rdgp** are identical to **ing** and **rdg** except they do not block. If they do not find a matching tuple, ε is returned. All of these operations are performed atomically, which can be guaranteed without a costly transaction because they are performed locally on a single agent.

[1]Both `valEql(`p`)` and `defaultConst(`p`)` are constraint functions that determine whether a tuple's field satisfies the template's constraint. In this case, `valEql(`p`)` returns *true* if the value within f is equal to p while `defaultConst(`p`)` always returns p.

INTERFACE SPECIFICATION:

void out(Tuple t)
— Places a tuple, t, into the tuple space.

Tuple rd(Template template)
— Blocks until a tuple matching the template is found within the tuple space. Returns a copy when found.

Tuple rdp(Template template)
— Returns a tuple from within the tuple space that matches the template, or ε if none is found.

Tuple[] rdg(Template template)
— Blocks until a tuple matching the template is found within the tuple space. When this occurs, a copy of all matching tuples are returned.

Tuple[] rdgp(Template template)
— Returns all tuples from within the tuple space that match the template, or ε none is found.

Tuple in(Template template)
— Blocks until a tuple matching the template is found within the tuple space. When this occurs, the tuple is removed and returned.

Tuple inp(Template template)
— Removes and returns a tuple from within the tuple space that matches the template, or ε if none is found.

Tuple[] ing(Template template)
— Blocks until a tuple matching the template is found within the tuple space. When this occurs, all matching tuples are removed and returned.

Tuple[] ingp(Template template)
— Removes and returns all tuples from within the tuple space that match the template, or ε if none is found.

Figure 5: Operations on the local tuple space.

INTERFACE SPECIFICATION:

[NOTE THAT THESE ARE ONLY REQUESTS]

void out(AgentLocation loc, Tuple t)
— Asks the agent located at **loc** to place a tuple in its tuple space.

Tuple rdp(AgentLocation loc, Template template)
— Returns a tuple matching the template from within the tuple space of the agent located at **loc**, or ε if none is found or the operation times out.

Tuple[] rdgp(AgentLocation loc, Template template)
— Returns all tuples matching the template from within the tuple space of the agent located at **loc**, or ε if none is found or the operation times out.

Tuple inp(AgentLocation loc, Template template)
— Removes and returns a tuple matching the template from within the tuple space of the agent located at **loc**, or ε if none is found or the operation times out.

Tuple[] ingp(AgentLocation loc, Template template)
— Removes and returns all tuples matching the template from within the tuple space of the agent located at **loc**, or ε if none is found or the operation times out.

Figure 6: Operations on a remote tuple space.

Remote LTS Operations. To allow for inter-agent coordination, agents allow remote agents to request that certain operation be performed on their tuple space. To do this, Limone provides operations **out**, **inp**, **rdp**, **ingp**, and **rdgp**, as shown in Figure 6. These methods differ from the local operations in that they require an AgentLocation parameter that specifies which agent should perform the operation. When one of these operations are executed, the agent on which it is executed sends a request to the remote agent specified by the AgentLocation, sets a timer, and remains blocked till either a response is received or the timer times out. When the remote agent receives the request, it passes it to the operation manager, which may reject or approve it. If rejected, an exception is returned to allow the initiating agent to distinguish between a rejection and a communication breakdown. If accepted, the operation is performed atomically on the remote agent, and the results are sent back to the initiating agent. The timer is necessary to prevent deadlock due to message loss. If the request or response is lost, the remote LTS operation will time-out and return ε. To resolve the case when an operation times out while the response is still in transit, the initiating agent enumerates each request, and the remote agent includes this value in its response.

Reaction Mechanism. Limone *reactions* enable an agent to inform other agents within its acquaintance list that it is interested in tuples that match a particular template. A reaction contains a user-defined call-back function that is executed by the agent that created it when a tuple of interest appears in a tuple space it is registered on. Reactions fit particularly well with ad hoc networks because they provide an asynchronous form of communication between agents by transferring the responsibility of searching for a tuple from one agent to another.

A reaction consists of a *reactive pattern* and a *call-back function*. The reactive pattern contains a template that indicates which tuples trigger it and a list of profile selectors that determine which agents the reaction applies to. The call-back function executes when the reaction *fires* in response to the presence of a tuple matching its template within the LTS it is registered on. The firing of a reaction consists of sending back to the issuing agent a copy of the tuple that triggered the reaction. Since message loss can occur at any time, the message sent to the issuing agent may be lost, meaning there is no guarantee that a reaction will fire even if a tuple matching the reactive pattern is found. If the issuing agent receives the message tuple, it will execute the reaction's call-back function. To prevent deadlock, the call-back function cannot perform blocking operations. If the call-back function were allowed to block, it would remain blocked forever because the call-back function is by definition atomic.

The list of profile selectors within the reactive pattern determines where to register (i.e., *propagate*) the reactive pattern. Implementation-wise, a profile selector is a template while a profile is a tuple. They are subject to the same pattern matching mechanism but are functionally different because profiles are not placed in tuple spaces. A reaction's reactive pattern propagates to a remote agent if the remote agent's profile matches *any* of the reactive pattern's profile selectors. Multiple profile selectors are used to lend the developer greater flexibility in specifying a reaction's domain. For example, returning to our example scenario, a delivery agent would have the following profile:

$$\textit{profile}\{\langle \text{“}\textit{type}\text{”},\ \textit{String},\ \text{“}\textit{Delivery Agent}\text{”}\rangle,\ \langle \text{“}\textit{location}\text{”},\ \textit{GPSCoord},\ (90.45\text{N},\ 34.23\text{W})\rangle\}$$

and its reactive pattern would contain the following profile selector to restrict its propagation to delivery agents:

$$\textit{profile selector}\{\langle \text{“}\textit{type}\text{”},\ \textit{String},\ \texttt{valEql}(\text{“}\textit{Delivery Agent}\text{”})\rangle\}$$

ABSTRACT STATE:

— A set of reactions, $\{r, \ldots\}$

INTERFACE SPECIFICATION:

ReactionID addReaction(Reaction rxn)
— Adds a reaction to the reaction registry and returns the reaction's ReactionID.

Reaction removeReaction(ReactionID rID)
— Removes and returns the reaction with the specified ReactionID from the reaction registry or ε if no reaction matching the ReactionID exists in the reaction registry.

Reaction get(ReactionID rID)
— Retrieves the reaction with the specified ReactionID from the reaction registry or ε if no reaction matching the ReactionID exists in the reaction registry.

Reaction get(Profile profile)
— Retrieves all reactions containing profiles that match the given profile or ε if no reaction matches.

Figure 7: Reaction Registry.

ABSTRACT STATE:

— A set of reactive patterns, $\{rp_1, rp_2, \ldots\}$

INTERFACE SPECIFICATION:

boolean addReactivePattern(ReactivePattern rp)
— Adds a reactive pattern to the reaction list, returns true if it was successfully added.

void clear()
— Clears the reaction list by removing all reactive patterns within it.

ReactivePattern[] getApplicablePatterns(Tuple tuple)
— Retrieves all of the reactive patterns within the list that should fire on the specified tuple.

void removeReactivePattern(ReactivePattern rp)
— Removes the specified reactive pattern from the list if it is in the list.

void removeReactivePatterns(AgentID aID)
— Removes all reactive patterns from the list that were registered by the agent with the specified AgentID.

Figure 8: Reaction List.

In this case the reactive pattern will propagate to any agent whose profile contains a property called "*type*," with a *String* value equal to "*Delivery Agent*." Notice that the profile selector did not consider the agent's location. This is because restrictions on the location of an agent can be done more elegantly using the engagement policy. If an agent wants to restrict reaction propagation to agents within 50m, it will set its engagement policy such that all agents within its acquaintance list are located within 50m.

Reactions may be of two types: ONCE or ONCE_PER_TUPLE. The type of the reaction determines how long it remains active once registered on a tuple space. A ONCE reaction fires a single time on each tuple space it is registered on and automatically deregisters itself after firing. When a ONCE reaction fires and the reference agent receives the resulting tuple(s), it deregisters the reaction from all other agents, preventing the reaction from firing later. If a ONCE reaction fires several times simultaneously on different tuple spaces, the reference agent chooses one of the results non-deterministically and discards the rest. This does not result in data loss because no tuples were removed from any tuple space. In contrast to ONCE reactions, ONCE_PER_TUPLE reactions remain registered after firing, thus firing once for each matching tuple. ONCE_PER_TUPLE reactions are deregistered at the agent's request or when network connectivity to the agent is lost. To keep Limone as lightweight as possible, no history is maintained regarding where reactions were registered. Thus, if network connectivity breaks and later reforms, the formerly registered reactions will be re-registered and will fire again.

Two additional state components, the *reaction registry* and *reaction list,* are required for the reaction mechanism. The reaction registry, shown in Figure 7, holds all reactions created and registered by the reference agent. An agent uses its reaction registry to determine which reactions should be propagated following an engagement and to obtain a reaction's call-back function when it fires.

The reaction list, shown in Figure 8, contains the reactive patterns registered on the reference agent's tuple space. The reactive patterns within this list may come from *any* agent within communication range, including agents *not* in the acquaintance list. Thus, to maintain the validity of the reaction list, the acquaintance handler notifies its agent when *any* agent moves out of communication range, not just the agents within its acquaintance list. The reaction list determines which reactions should fire when a tuple is placed into the local tuple space or when a reactive pattern is added.

A simple illustration of how the geocast example could be implemented is given in Figure 9. The agent's constructor creates and registers a reaction that is sensitive to message tuples. Since the reaction is created using an empty `ProfileSelector`, it is propagated to all agents in the acquaintance list, which the engagement policy limits to other delivery agents. The call-back function of the reaction is defined in the `reactsTo` method. It determines whether to pull or place the tuple into its tuple space based on the destination of the message as specified within the tuple.

Agent Mobility. Coordination within Limone is based on the logical mobility of agents and physical mobility of hosts. Agents are logically mobile since they can migrate from one host to another. Agent mobility is accomplished using a package called μCode [7]. μCode provides primitives to support light-weight mobility preserving code and state. Of particular interest is the μCodeServer and mobile agent. A mobile agent maintains a reference to a μCodeServer and provides a `go(String` *destination*`)` method that moves the agent's code and variable state to the destination. The thread state of the agent is not preserved because doing so would require modification to the Java virtual machine, limiting Limone to proprietary interpreters. Thus, after an agent migrates to a new host, it will start fresh with its variables initialized to the values they were prior to migration.

Limone cooperates with μCode by running a μCodeServer alongside each *Limone* Server and having the *Limone* agent extend μAgent. By extending μAgent, the Limone agent inherits the `go(String` *destination*`)` method. However, Limone abstracts this into a `migrate(HostID` hID`)` method that moves the agent to the destination host by translating the `HostID` to the string accepted by μCode. Just prior to migration, the agent first deregisters all of its reactive patterns from remote agents, and stops its beaconing. By not

```
public class DeliveryAgent extends Agent implements ReactionListener {
    public DeliveryAgent (AgentID aID) {
        EConstraint c1 = new EConstraint("type", String.class,
            new EquivalencyConstraintFunction("Directed Multicast"));
        EConstraint c2 = new EConstraint("message", String.class,
            new DefaultConstraintFunction());
        EConstraint c3 = new EConstraint("destination", GPSCoord,
            new DefaultConstraintFunction());
        ETemplate template = new ETemplate();
        template.addConstraint(c1).addConstraint(c2).addConstraint(c3);
        ReactivePattern rPat = new ReactivePattern(new ProfileSelector(),
            Reaction.ONCE_PER_TUPLE, template);
        Reaction rxn = new Reaction(rPat, this); // create the reaction
        ReactionID rID = null;
        try {
            reactionRegistry.registerReaction(rxn); // register the reaction
        } catch(TupleSpaceException e) { e.printStackTrace(); }
    }
    public void reactsTo(ReactionEvent e) { // the call-back function
        Tuple msg = e.getTuple();
        GPSCoord dest = (GPSCoord)msg.getField("Destination").getValue();
        if (isToMe(dest)) tupleSpace.out(msg);
        else{
            AgentLocation cLoc
                = getLoc(msg.getField("AgentID").getValue());
            if (iAmCloser(dest, cLoc)) {
                Tuple grabbed = tupleSpace.inp(aLoc,msg.getTemplate());
                if (grabbed != null) tupleSpace.out(grabbed);
            }
        }
    }
}
```

Figure 9: Example implementation of a delivery agent. Due to space constraints, the setting of the engagement policy is not shown. In the actual implementation, a `ProfileSelector` would be created and used to limit engagement only with other DeliveryAgents.

broadcasting beacons, neighboring agents will assume the migrating agent has moved out of range and thus disengage with it. Once on the new host, the agent is passed to the Limone Server which restarts it and resumes the broadcasting of its beacons. This allows remote agents to re-engage with the agent at its new location.

5. DISCUSSION

A prototype implementation of Limone has been developed using Java. The prototype strictly adheres to the model given in Section 2, where each construct is a distinct object that implements the interface and behavior described in Section 4.

For a host to participate in Limone, it must create and activate a Limone Server. When a Limone Server is created it is initially inactive, does not open any ports, and does not support any agents. The application activates the Limone Server by calling `boot()` on it. Prior to booting the server, the application may customize various parameters of the server such as the ports used, its multicast group address, beacon broadcast period, and even the single-cast protocol used (either TCP or UDP). Allowing the Limone Server to use either protocol makes it more scalable to small devices that cannot support the overhead of TCP or applications that do not require the additional delivery guarantees that TCP provides. The limitation is that a Limone Server can only communicate with other Limone Servers that use the same single-cast protocol. When booted, the Limone Server opens and listens to a single cast port for incoming messages and starts broadcasting beacons on the multicast port. For efficiency purposes, broadcasting of beacons is relegated to the Limone Server instead of to individual agents. Thus, each beacon contains a profile for each agent residing on the server. Even if no agents are on the server, it must still broadcast beacons for its presence to be known to agents residing on neighboring servers.

Once a Limone Server has been created and booted, the application can load agents onto the server. This can either be done by calling `loadAgent(...)` on the Limone Server, or by using a special `Launcher` object that communicates to the server through its single-cast port. The `Launcher` allows new agents to be loaded onto the Limone Server at any time, possibly from a remote device.

Limone provides a default implementation of an Agent. An application can interact with an agent either directly by passing the agent a reference to it upon creation, or by subclassing the agent and overriding the agent's methods to include the behavior it desires.

As a testament to how lightweight Limone is, its jar file is 111.7KB, compared to LIME's 655.8KB jar file. To analyze the performance of Limone, we calculated the round trip time for a tuple containing eight bytes of data to be pulled onto a remote agent and back using reactions as triggers. The experiment is as follows: Given two agents, A and B, A has a global reaction registered for red tuples, while B has a global reaction registered for green tuples. A places a green tuple into its tuple space. When agent B reacts to

Model	Lines of Code	Time (ms)
Limone	250	50.3
LIME	170	73.6
Raw Sockets	695	44.6

Figure 10: Application code size and round-trip message passing time using reactions as a trigger, averaged over 100 rounds.

it, it places a red tuple into its tuple space. Both types of tuples carry eight bytes of data. The actual time measured begins at the insertion of the green tuple by A to the firing of its reaction. The test was performed using two 750MHz laptops running Java 1.4.1 in 802.11b ad hoc mode with a one second beaconing period. The laptops were located in a "clean room" environment where the laptops are stationary and sitting next to each other. To compare Limone's performance, we also performed the same operation using LIME and raw TCP sockets. Averaged over 100 rounds, the results of our tests are shown in Figure 10. One can easily observe that, Limone adds some overhead over raw sockets, but not as much as LIME. Interestingly, while Limone decreases the amount of code the application developer must write, it still requires more code than LIME. This is due to Limone's more expressive pattern matching mechanism and engagement policy.

Limone assumes that the rate of configuration changes is small relative to the network latencies. This is a universal assumption made by all distributed applications. If configuration changes are so rapid that they exceed message latencies, then the majority of messages will be lost, making coordination activities impossible.

In a wireless ad hoc network, it is possible for different devices to have different broadcast ranges. If this occurs, it may be possible for an agent to discover another agent, but not be able to send it messages because of its limited broadcast range. This anomaly is elegantly addressed by Limone since the remote agent's profile will be automatically removed upon communication failure.

6. RELATED WORK

This section explores the expressive power of Limone by comparing it to several other coordination models and, when possible, demonstrating how Limone can offer identical or similar support for key concepts and constructs. The models considered here include JEDI [8], LIME [1], MARS [9], and PEERWARE [10].

JEDI. JEDI is a model based on the event subscription paradigm where components create and subscribe to events. It consists of active objects that interact with each other through a logically centralized event dispatcher. Active objects subscribe to, or unsubscribe from, events on the event dispatcher. When an active object registers an event on the event dispatcher, it gives the event dispatcher an event that it passes to all active objects to which it subscribes. This provides a powerful decoupling among the active objects (i.e., the active object that created the event need not know which active objects received it). Logical mobility is possible in JEDI since active objects can unsubscribe from an event dispatcher on one host, and resubscribe to another one on another host.

The behavior of JEDI's event subscription mechanism can be captured in Limone through reactions that apply to all agents in the acquaintance list. These reactions would be sensitive to special event tuples. JEDI events can be represented in Limone using these event tuples.

Lime. LIME is another coordination model implemented as middleware for mobile environments. LIME, which preceded Limone, supports ad hoc networks, utilizes logically mobile agents running on physically mobile hosts, and coordinates through Linda-like tuple spaces enhanced with reactive programming. Unlike Limone which was designed to be as light-weight as possible, LIME provides strong atomicity and functional guarantees. LIME follows a transactional paradigm where operations often occur as a single atomic transaction. For example, when two groups of hosts merge, the engagement and reaction propagation is done between all hosts as a single atomic step through a distributed transaction. This level of atomicity comes at a cost. Since it requires every host to send a message to every other host, the amount of unnecessary message-passing is higher. It also requires all hosts to remain in contact with each other throughout the transaction, which may be difficult to guarantee, particularly in highly dynamic environments with a high density of hosts. In contrast, Limone follows an incremental paradigm where engagements between two groups of agents are performed gradually by each agent independently. Once an agent engages with another agent, reaction propagation follows suit in a similar gradual manner.

A key difference between LIME and Limone is the engagement policy and the number of tuple spaces used. LIME's engagement policy is symmetric and built into the model whereas Limone's policy is application-customizable and asymmetric. In Limone each agent has an individual tuple space whereas in LIME all agents on a host share multiple host-level tuple spaces that are differentiated by name. When a group of hosts forms in LIME, their identically named tuple spaces merge into one in a single atomic step. Using a single tuple space simplifies Limone without reducing its functionality since multiple tuple spaces can be simulated using a field within each tuple to identify which "virtual" tuple space it belongs to.

Due to fundamental differences between the two models, Limone cannot easily provide the level of atomicity guarantees that LIME provides. However, it can provide the general functionality of LIME's distributed operations with relaxed atomicity guarantees. For example, LIME provides a global **in** operation that atomically searches the tuple space on all hosts within a group. It guarantees that if a matching tuple exists, it will be found. Although Limone cannot provide such a guarantee, it can sequentially perform an **inp** operation on each acquaintance until it finds a match. While this does not guarantee the match will be found, the probability of success is high. This reflects the highly pragmatic oriented approach the design of Limone has followed.

MARS. MARS consists of a multiplicity of *nodes* each containing a tuple space. Agents located on a particular node coordinate by placing tuples into and removing tuples from the node's tuple space and a reaction mechanism sensitive to actions performed by an agent. Since agents can only access their node's tuple space, they can only coordinate with other agents located on the same node. Migration is required for inter-node communication. MARS adapts to mobile environments by allowing agents to "catch" connec-

tion events that indicate the presence of a remote node, to which they may migrate.

The general behavior of a MARS node can be achieved in Limone by restricting agents to engage only with agents on the same host and configuring the operation manager to reject requests coming from agents residing on another host. In this case, a MARS node is essentially a Limone host. The difference is tuples remain associated with a particular agent, and move with it during migration. Limone's reaction mechanism can also be arranged to behave like those in MARS. In MARS, a reaction fires due to an operation being performed. A Limone reaction can behave like a MARS reaction by having the agent insert special "event tuples" each time it performs an operation on the tuple space and configuring the reaction to react to these tuples.

PeerWare. PEERWARE is primarily concerned with the creation and maintenance of a virtual tree data structure that is built by virtual superimposition of numerous local trees. The use of a tree helps PEERWARE scale to large data sets since, when looking for data in a particular branch, not all of the data has to be searched. Each data object (or node) in a local tree is named. Multiple nodes within a tree can have the same name as long as they are not part of the same branch and are not roots. The local trees are superimposed upon each other based on the names of the nodes. Changes in network configuration are represented as changes in the global tree's content. The operations that can be performed on the global tree are similar to those allowed on the tuple space (e.g., data insertion and extraction). Like Limone, PEERWARE does not provide any atomicity guarantees on distributed operations but does guarantee that they will execute atomically at the local level. PEERWARE provides an `execute` function that performs a user-defined operation on a projection of the tree. This is useful especially when the operation is relatively small and accesses large data sets since the data does not need to be sent over the network.

The behavior of PEERWARE can be accomplished using Limone by adding special application-defined fields into each tuple to indicate where it belongs in the tree. The fields could then be used by an application to simulate the scoping properties of a tree. Although this is less efficient, specialized implementations of a specific data structure will always be more efficient. PEERWARE's `execute` function is provided in Limone by creating an agent that performs the desired operation, and migrating it to the remote host to perform the operation.

7. CONCLUSIONS

Limone is a lightweight but highly expressive coordination model and middleware tailored to meet the needs of developers concerned with mobile applications over ad hoc networks. Central to Limone's function is the management of context-awareness in a highly dynamic setting. At first glance, an agent's context is a subset of the agents in direct contact as they appear in the acquaintance list. At this level, the context is transparently managed and subject to policies imposed by each agent in response to its own needs at a particular point in time. Explicit manipulation of the context is provided by operations that access data owned by agents in the acquaintance list. The agent retains full control of the local tuple space since all remote operations are simply requests to perform a particular operation for a remote agent and are subject to policies specified by the operation manager. This high degree of security encourages a collaborative type of interaction among agents. An innovative adaptation of the reaction construct facilitates rapid response to environmental changes. As supported by evidence to date, the result of this unique combination of context management features is a coordination model and middleware that promise to reduce development time for mobile applications.

Acknowledgements This research was supported in part by the National Science Foundation under grant No. CCR-9970939 and the Office of Naval Research under MURI Research Contract N00014-02-1-0715. 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 research sponsors.

8. REFERENCES

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

[2] Picco, G., Murphy, A., Roman, G.C.: LIME: Linda meets mobility. In: Proc. of the 21^{st} Int'l. Conf. on Software Engineering. (1999)

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

[4] Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proc. of the 10^{th} Int'l. Symp. on Foundations of Software Engineering. (2002)

[5] McCann, P.J., Roman, G.C.: Compositional programming abstractions for mobile computing. IEEE Transactions on Software Engineering **24** (1998) 97–110

[6] Navas, J.C., Imielinski, T.: Geocast - geographic addressing and routing. In: Proceedings of the Third Annual International Conference on Mobile Computing and Networking. (1997) 66–76

[7] Picco, G.P.: code: A lightweight and flexible mobile code toolkit. In Rothermel, K., Hohl, F., eds.: Proceedings of the 2nd International Workshop on Mobile Agents. Lecture Notes in Computer Science, Berlin, Germany, Springer-Verlag (1998) 160–171

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

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

[10] Cugola, G., Picco, G.: Peerware: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano (2001)