Report Number: WUCSE-2002-47

2002-12-30

A Lightweight Coordination Model and Middleware for Mobile Computing **Please see WUCSE-03-12**

Gruia-Catalin Roman and Chien-Liang Fok

LimeLite is a new coordination model and middleware designed to support rapid development of applications entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, LimeLite performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies specified at the application level. This asymmetry among participants in the coordination process is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. It represents an important departure from coordination research in general. The coordination... **Read complete abstract on page 2.**

A Lightweight Coordination Model and Middleware for Mobile Computing **Please see WUCSE-03-12**

Gruia-Catalin Roman and Chien-Liang Fok

Complete Abstract:

LimeLite is a new coordination model and middleware designed to support rapid development of applications entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, LimeLite performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies specified at the application level. This asymmetry among participants in the coordination process is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. It represents an important departure from coordination research in general. 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. Linda-like primitives typically used in coordination middleware are tailored in LimeLite to address the challenges of mobile environments. Among other things, this entails the elimination of remote blocking and data pushing operations since the affected agents may no longer be within communication range. It also entails the addition of reactions that are triggered by the presence of information of interest on agents listed in the acquaintance list and not by events that could have occurred prior to discovery. Finally, to ensure both performance and ease of deployment on small devices the granularity of atomic operations and the reliance on transport layer guarantees have been minimized. This paper introduces LimeLite, explains its key features, illustrates its usage in application development, and explores its effectiveness as a software engineering tool.

2002-47

A Lightweight Coordination Model and Middleware for Mobile Computing **PLEASE SEE WUCSE-03-12**

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

December 30, 2002

Abstract: LIMELite is a new coordination model and middleware designed to support rapid development of applications entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, LIMELite performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies specified at the application level. This asymmetry among participants in the coordination process is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. It represents an important departure from coordination research in general. 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. Linda-like primitives typically used in coordination middleware are tailored in LIMELite to address the challenges of mobile environments. Among

A Lightweight Coordination Model and Middleware for Mobile Computing

Gruia-Catalin Roman and Chien-Liang Fok

Mobile Computing Laboratory
Department of Computer Science and Engineering
Washington University
Saint Louis, Missouri 63130-4899, USA
{roman, liang}@cse.wustl.edu
http://mobilab.cse.wustl.edu

Abstract. LimeLite is a new coordination model and middleware designed to support rapid development of applications entailing logical mobility of agents and physical mobility of hosts. Designed to function in open environments, LimeLite performs automatic agent discovery but filters the results to define for each agent an individualized acquaintance list in accordance with run-time policies specified at the application level. This asymmetry among participants in the coordination process is dictated by the need to accommodate settings involving large numbers of agents and hosts that come and go freely. It represents an important departure from coordination research in general. 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. Linda-like primitives typically used in coordination middleware are tailored in LimeLite to address the challenges of mobile environments. Among other things, this entails the elimination of remote blocking and data pushing operations since the affected agents may no longer be within communication range. It also entails the addition of reactions that are triggered by the presence of information of interest on agents listed in the acquaintance list and not by events that could have occurred prior to discovery. Finally, to ensure both performance and ease of deployment on small devices the granularity of atomic operations and the reliance on transport layer guarantees have been minimized. This paper introduces LimeLite, explains its key features, illustrates its usage in application development, 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 grow quickly 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 may belong to the enemy. Ad hoc networks are also convenient in day-to-day scenarios where the duration of the activity is simply too brisk and too 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. Mobile applications must be robust enough to handle the possibility of disconnection at any point in time. The physical size and power consumption of mobile devices limits their functionality and further exacerbates the difficulties associated with meeting application demands. The limited functionality of mobile devices often leads to strong mutual dependencies. Devices may not be able to function fully in isolation, resulting in a greater need for coordination support. For example, in a planetary exploration setting ad hoc networking enables miniature rovers each equipped with only a few specialized sensors to carry out experiments that demand data from an arbitrary combinations of sensors.

Mechanisms that address the complexities introduced by mobility include enhancements to the operating system, specialized languages, and middleware. Among these approaches, middleware has emerged as the most popular. Operating systems are often tightly integrated with low-level communication services (e.g., TCP sockets) and, as such, they are likely to 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 thus entail too high a risk. Middleware, on the other hand, provides higher level abstractions while minimizing risk by taking advantage of existing software infrastructure. Within the context of established languages, middleware provides higher levels of abstraction than that of the operating system. When designed properly, it can free developers from dealing with mundane areas that have already been well investigated such as the protocol layer and allow them to focus on more fruitful topics like models, algorithms, and applications.

Several coordination models for mobile environments have been developed. These models include LIME [1, 2], MARS [3], and PeerWare [4]. To the best of our knowledge, LIME is the only model to support ad hoc mobility. It is a coordination model that uses distributed transactions to process configuration changes and assumes all unannounced disconnections are fully masked. The heavy-weight nature of LIME makes it unsuitable for devices with limited resources. Fortunately, many applications for ad hoc networks do not require the level of atomicity guarantees that LIME provides. It is this particular observation that motivated this research effort.

In this paper we introduce LIMELite, a new lightweight coordination model and middleware for mobile environments supporting logical mobility of agents

and physical mobility of hosts. LimeLite 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. The acquaintance list is introduced as a new abstraction which defines a personalized view of the operation context. For each agent, LimeLite automatically performs remote agent discovery and maintains an acquaintance list of agents within communication range by using policies specified at the application level and subject to continuous revision. As in most coordination models, traditional Linda-like primitives over tuple spaces [5] facilitate the coordination of agent activities. However, LimeLite provides advanced pattern matching capabilities, allows agents to restrict the scope of their operations solely to agents satisfying specific application-defined eligibility criteria, and offers a powerful repertoire of reactive programming constructs. The autonomy of each agent is maintained by the explicit exclusion of remote blocking operations, group transactions, and data pushing primitives. Furthermore, LimeLite ensures that all remote operations are done between two agents with built-in mechanisms to address the possibility of data loss or disconnection. By emphasizing minimality of concepts, simplicity and feasibility of implementation, LimeLite operations are (by and large) resilient to message loss and unexpected disconnection. This allows LimeLite to function in ad hoc environments where existing models cannot.

The paper starts with an overview of the LimeLite model in Section 2. Section 3 presents a motivational example that describes how an application providing spatially-directed multicasting can be implemented using LimeLite. Following the example, Section 4 presents the run-time environment provided by LimeLite. This section describes the functionality of the constructs provided to the application. We then proceed with a discussion in Section 5 of the assumptions and vulnerabilities in LimeLite, and elaborate on the design of the LimeLite middleware. We end with a section on related work (Section 6) that describes how LimeLite can be used to implement existing coordination models, and draw conclusions in Section 7.

2 Model Overview

LimeLite assumes a computational model consisting of mobile devices (hosts) that form ad hoc networks; mobile agents that reside on hosts and may migrate from one host to another; and data owned by agents that is shared through a distributed Linda-like tuple space [5]. The relationship between hosts and agents is shown in Figure 1. Secure transmission, agent and host authentication, and data transport mechanics are assumed to be available to the implementation layer and thus absent from the logical view of the model.

The features of LimeLite 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 to the current task.

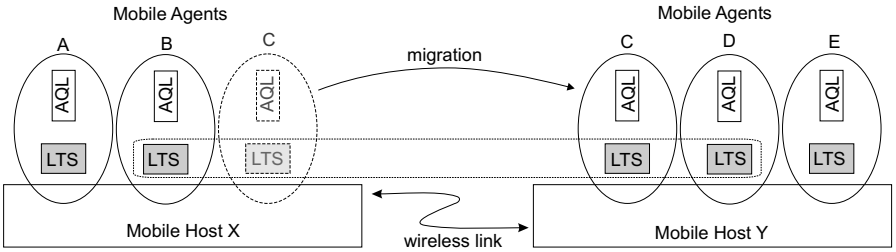

Fig. 1. An overview of the LimeLite model. Agents are represented as ovals. Each agent owns a local tuple space (LTS) and an acquaintance list (AQL). In this example, agent C is shown as migrating to host Y without a change in its acquaintance list, which consists of B and D. The dotted rectangle surrounding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

LimeLite provides a discovery protocol that informs each agent of the arrival and departure of other agents. It notifies each agent of its relevant neighbors by storing them in individualized acquaintance lists, where the relevance is determined using a filter (known as an *engagement policy*) specified at the application level. Since each agent has different neighbors and individualized engagement policies, the context perceived by each agent is generally different from that of its peers. This asymmetry among agents was first introduced in EgoSpaces [6]. It increases the level of decoupling among agents and results in a more robust coordination model that requires fewer assumptions about the underlying transport layer.

Existing coordination models for mobility in ad hoc environments such as Lime presume a symmetric and transitive coordination relation among agents that is not scalable. If every node must coordinate with every other node, the 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 only to agents it is interested in, LimeLite 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 remote agents, it can concentrate on these two agents by ignoring the rest, thus minimizing wasted memory and other resources.

LimeLite accomplishes explicit data access in a manner similar to that employed by most other coordination models. Each agent owns a tuple space which offers operations for placing tuples into it and for retrieving tuples through a pattern-matching mechanism related to that used in Linda. The use of a separate tuple space within each agent is not limiting; LimeLite can mimic the behavior of multiple tuple spaces á la Lime by utilizing special fields within each tuple, and can mimic a single shared tuple space per host á la MARS by setting the engagement policy to consider only agents on the local host.

Explicit data access spans at most two agents. The agent initiating the data access, referred to as the reference agent, must have the other agent in its ac-

quaintance list. LimeLite adopts a pull-only paradigm where the reference agent can read or remove data from other agents in its acquaintance list, but cannot push data onto them. The rationale is that most mobile applications involve agents pulling resources from their environment as needed. This design philosophy also protects valuable local resources (e.g., memories) from being consumed by the careless behavior of other agents. Finally, since disconnection may occur at any time, the probability of data loss is minimized.

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, LimeLite automatically propagates the reaction to all agents in the acquaintance list that satisfy certain properties specified by the application (e.g., agent location). At the receiving end, the reaction list monitors which reactions are registered on the local tuple space. When a tuple in the tuple space satisfies the trigger for a reaction in the reaction list, the agent that registered the reaction is notified and a copy of the tuple is sent. If this agent is still within range and receives the notification, it executes the code associated with the reaction locally. This mechanism, originally introduced in Mobile UNITY [7], is distinct from that employed in traditional publish/subscribe systems. It is designed to react 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 LimeLite by allowing agents to migrate from one host to another. When an agent migrates, LimeLite automatically updates its context and reactions. There are many benefits to allowing an agent to migrate. For instance, if a particular host has a large amount of data which it is not willing to give up, 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. As another example of agent mobility, suppose one 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, 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 web. 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 LimeLite by focusing on a simple problem involving a geocast [8]. 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

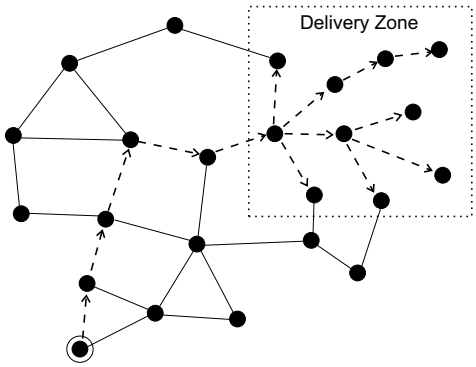

Fig. 2. This figure shows the topology of an ad hoc network and an example geocast application. For illustration purposes, the topology is fixed. Each dot is an agent. In this example, the circled agent sends a message to all agents encompassed within the rectangle. The arrows depict the path by which 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.

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.

The scenario just described can be easily implemented using LimeLite through 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 own tuple space. Special "delivery" agents that have been deployed on each host have reactions sensitive to this message 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, the message's present location, and its destination location. If the delivery agent is located closer to the destination than the message is, it will pull the tuple containing the message 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 during message delivery). 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, meaning only one agent will successfully grab the tuple. When the message reaches a destination agent, it can place the tuple into its own tuple space, resulting in the reactions of other destination agents to fire. 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. 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 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 is given in Section 4. An analysis of its performance is by far more complex than its coding and is outside the scope of this section. This is exactly what we should expect in an unpredictable ad hoc setting when the right resources are provided to the application programmer. At this point, the message we want to leave the reader with is that LimeLite has the potential to significantly reduce development efforts in a mobile setting.

4 Run-Time Environment

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

An application uses LimeLite by interacting with an agent, or by creating agents that perform the duties of the application. Each agent contains a tuple space, acquaintance list, reaction registry, and reaction list. The overall structure of LimeLite is shown in Figure 3. An agent allows the application to customize its profile and engagement policy. 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. This section describes how LimeLite fulfills its responsibilities and is organized around the key elements of the run-time environment, i.e., agent discovery, management, reactions, and agent mobility.

Discovery Mechanism. Since network connectivity between hosts in ad hoc networks can form and break at any time, LimeLite 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 construct in LimeLite 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 LimeLite server. A profile is a collection of triples each consisting of a property name, type, and value. Each profile always contains two system-defined entries indicating the agent location and identifier in addition to entries reflecting various application-defined char-

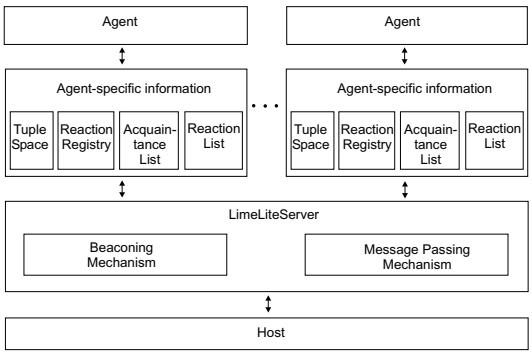

Fig. 3. The overall structure of LimeLite.

acteristics. When the LimeLite server receives a beacon, it forwards a copy to each agent running on top of it.

Upon receiving a beacon, an agent passes the profiles within it to its *acquaintance handler,* which enforces the *engagement policy* (a user-definable set of restrictions over profiles). For example, an engagement policy may contain no restrictions at all, or restrict engagement to agents located within a particular area. The acquaintance handler determines whether any new agents have either entered or left communication range. When the acquaintance handler receives a set of profiles, it determines whether any of the profiles represent agents not in the reference agent's acquaintance list. If some do, the engagement policy decides whether to include them in the acquaintance list. The reference agent is engaged with all agents in its acquaintance list.

It is also possible for an agent to be removed from the acquaintance list when its profile no longer satisfies the engagement policy. This is called a disengagement. To handle network disconnections, the acquaintance handler keeps track of the most recent update for all known agent profiles. Profiles that are too old, according to some application-defined threshold are dropped and removed 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 not. 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

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(Profile[] profiles)
— Returns all of the profiles within the list that match any of the specified profiles.
void remove(Profile profile)
— Removes the specified profile from the list.

Fig. 4. Acquaintance list.

is performed as a single atomic transaction, which is possible to do inexpensively because it is performed locally.

Tuple Space Management. Any data available for coordination among agents is stored in individually owned tuple spaces. Each contains a set of tuples each having a unique identifier. LimeLite 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 LimeLite by unordered collections of named fields. This results in a more powerful 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 a message tuple used in the earlier example:

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

Agents use templates to specify tuples of interest in the tuple space. A template consists of a collection of named constraints, each defined in terms of the field name to which it applies and a predicate over the field type and value. Because of the manner by which the predicate is supplied, it is 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:

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

[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`)`

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 if 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.

Fig. 5. Operations on the local tuple space.

The bottom two constraints with default constraint functions that always return *true* specifies that all matching tuples must contain fields with the specified names and types (i.e., it must have a field named "*message*" with a value of type *String* and a field named "*destination*" with a value of type *GPSCoord*). 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 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

returns *true* if the value within f is equal to p while `defaultConst(`p`)` always returns p.

INTERFACE SPECIFICATION:

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.

Fig. 6. Operations on a remote tuple space.

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.

Remote LTS Operations. To allow for inter-agent coordination, agents share the contents of their tuple spaces with other agents. To share the contents of the tuple space, LimeLite provides operations **inp**, **rdp**, **ingp**, and **rdgp**, as shown in Figure 6, that operate on the tuple spaces of remote agents. These methods differ from the local operations in that they require an AgentLocation parameter that specifies on which agent's tuple space the operation should be performed. Despite the distributed nature of these operations, the actual querying of the tuple space is still performed atomically. All of these operations are implemented using message passing, which is hidden from the agent. Due to the possibility of message loss, these operations are not guaranteed to return a matching tuple even if one exists. To prevent deadlock due to lost messages, all of these operations will time-out and return ε if results are not received after a certain system-defined period of time.

No operations on remote tuple spaces can block because the system could deadlock if the network connection were to break in the middle of the operation. A drawback of not having blocking operations on remote agents is the need for polling to detect the presence or absence of a tuple in the remote agent's tuple space. Given the bandwidth limitations of wireless networks and battery constraints of mobile devices, polling should be avoided. LimeLite avoids this by providing a reaction mechanism.

Reaction Mechanism. LimeLite *reactions* enable an agent to inform other agents within its acquaintance list that it is interested in tuples that match a particular template. Reactions are *registered* and *deregistered* on tuple spaces. 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 agent's tuple spaces it should be registered on. The call-back function executes when the reaction *fires* in response to the existence 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, followed by the execution of the reaction's call-back function on the issuing agent. To prevent deadlock, the call-back function cannot perform blocking operations. If the call-back function were allowed to block on the local tuple space, it will remain blocked forever because the call-back function is atomic meaning no matching tuple can be placed into the tuple space to un-block it.

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 share 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. Returning to our example scenario, a delivery agent would have the following profile:

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

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

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

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 is done using the engagement policy. For example, 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

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.

Fig. 7. Reaction Registry.

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 found in each tuple space it is registered on. ONCE_PER_TUPLE reactions are deregistered when the agent requests it or when network connectivity to the agent is lost. To keep LimeLite as lightweight as possible, no history is maintained on 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 to it.

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

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.

Fig. 8. Reaction List.

or place the tuple into its tuple space based on the destination of the message as specified within the tuple.

Agent Mobility. Coordination within LimeLite is based on the logical mobility of agents and physical mobility of hosts. Agents are logically mobile in that they can migrate from one host to another throughout their lifetime. Agent mobility is accomplished using a package called μCode [9]. 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 LimeLite 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.

LimeLite cooperates with μCode by running a μCodeServer alongside each LimeLite Server and having the LimeLite agent extend μAgent. By extending μAgent, the LimeLite agent inherits the `go(String` *destination*`)` method. However, LimeLite 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 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 LimeLite Server which restarts it and resumes the broadcasting of its beacons.

```
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 {
            tupleSpace.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);
            }
        }
    }
}
```

Fig. 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`.

5 Discussion

A prototype implementation of LimeLite has been developed using Java. The prototype attempts to adhere to the model given in Section 2. The implementation defines the LimeLite Server, Agent, Tuple Space, Acquaintance List, Acquaintance Handler, Reaction List, Reaction Registry, Tuples, Templates, Profiles, and Profile Selectors as distinct objects. Each of these objects implement the interface and behavior as described in Section 4.

For a host to participate in LimeLite, it must create and activate a LimeLite Server. When a LimeLite Server is created it is initially inactive and does not open any ports nor support any agents. The application activates the LimeLite 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 LimeLite 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 LimeLite Server can only communicate with other LimeLite Servers that use the same single-cast protocol. When booted, the LimeLite Server opens and listens to a single cast port for incoming messages and starts broadcasting beacons from the multicast port. For efficiency purposes, broadcasting of beacons is relegated to the LimeLite Server instead of to individual agents. A 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. To allow for agent migration, the LimeLite Server allows an agent to indicate that it should no longer be included in the beacons. When the agent's profile is no longer included in the beacons, remote agents will assume the agent longer exists and disengage with it.

Once a LimeLite Server has been created and booted, the application can load agents onto the server. This can either be done by calling a `loadAgent(...)` method on the LimeLite 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 LimeLite Server at any time.

LimeLite provides a default implementation of an Agent that holds an acquaintance handler, acquaintance list, tuple space, reaction registry, and reaction list. 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.

The LimeLite JAR file is only 52.7KB in size. However, it uses several external packages that makes the total code size 111.7KB which may be too high considering current mobile devices are often limited to 300KB of memory. However, we believe this is not a problem because continuous improvements in technology will result in increasing memory capacity.

To analyze the performance of LimeLite, we calculated the round trip time for a tuple to be pulled onto a remote agent and back using reactions as triggers. 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. Whenever agent B reacts to a green tuple, 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 the reaction sensitive to the red tuple on A. The test was performed on two 750MHz SONY laptops running Java 1.4.1 in 802.11b ad hoc mode with a beaconing period of 1000ms. To judge the efficiency

of LimeLite, we compared the round-trip time of simple message passing using eight byte messages and TCP sockets on the same machines. Averaged over 100 rounds, the round trip time of LimeLite is 50.348ms while the plain sockets is 44.58ms. The amount of LimeLite code required was 250 lines totalling 8.18KB whereas the regular message passing took 695 lines and 19.19KB.

Although LimeLite as been designed with robustness and simplicity in mind, it, like all other coordination models, has certain vulnerabilities that arise in rare network situations. While none of these vulnerabilities will result in system failure, they may result in individual operations failing to deliver the expected results. These vulnerabilities are worth pointing out so that application developers will be aware of their possibility when designing LimeLite applications.

LimeLite makes two assumptions about the underlying network. The first assumption is that the rate of configuration changes is small relative to the network latencies. If configuration changes are so rapid that they exceed message latencies, then the majority of messages will be lost, making coordination impossible. If this occurs, LimeLite will not function correctly because the results of a remote **inp** or **ingp** may be lost during transmission.

LimeLite also assumes that the broadcast range of all devices is the same. If this is not the case, then it is possible for messages to be sent in one direction but not the other. Although this will not cause LimeLite to crash, it may result in inconsistencies between agents regarding the registration of reactions. The consequences can be minimized by assuming the range of all devices in the network to be a fraction of the shortest transmission range among all the devices.

Finally, other than the pull-only principle, LimeLite does not address the issue of access control. There is no policy for an agent to control who accesses its tuple space, or for which tuples can be removed. We believe that such measures can be readily introduced into LimeLite at a later date.

6 Related Work

This section explores the expressive power of LimeLite by comparing it to several other coordination models and, when possible, demonstrating how LimeLite can provide their basic concepts. The models considered include JEDI [10], Lime [1], MARS [3], and PeerWare [4].

JEDI. JEDI is a model based on the event subscription paradigm where components create and subscribe to events. JEDI 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 LimeLite 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 LimeLite using these event tuples.

Lime. Lime is another coordination model implemented as middleware for mobile environments. Like LimeLite, Lime 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 LimeLite which was designed to be as light-weight as possible, Lime is relatively heavy-weight providing strong atomicity and functional guarantees. LimeLite 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. In contrast, 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.

A key difference between Lime and LimeLite is the engagement policy and the number of tuple spaces used. Lime's engagement policy is symmetric and built into the model. LimeLite's policy is variable and asymmetric. In LimeLite 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. LimeLite does not provide multiple tuple spaces. Using a single tuple space simplifies the model without reducing its functionality since multiple tuple spaces can be simulated using a field within each tuple to identify which simulated tuple space it belongs to.

Due to fundamental differences between the two models, LimeLite 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 LimeLite 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.

MARS. MARS consists of a multiplicity of *nodes* each containing a programmable tuple space. Agents located on a node maintain a private reference to a tuple space that is bound to the tuple space of the node they are located on. As the agents migrate from one node to another, their tuple space reference

is automatically updated to point to the new tuple space of the new node. This is in contrast to LimeLite where each agent maintains its own tuple space and carries it as it migrates. Coordination between agents in MARS is done through placing and removing tuples in and out of the 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 same node. Migration is required for inter-node communication. MARS was initially designed to operate in a wired network environment. As such, it did not contain a discovery protocol. However, recent versions of MARS have been adapted to mobility by allowing mobile agents to "catch" connection events that occur when they come into range of a node. When this occurs, they may migrate to the newly discovered node.

The general behavior of a MARS node can be achieved in LimeLite by restricting each agent to only engage with agents on the same host. In this case, a MARS node is essentially a LimeLite host. The difference is that the tuples stored at a particular host remain associated with a particular agent, and move with the agent when it migrates to another host. LimeLite'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 LimeLite reaction that is sensitive to tuple space state 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 fire on 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 LimeLite, 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 LimeLite 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 LimeLite by creating an agent that performs the desired operation, and migrating it to the remote host to perform the operation.

7 Conclusions

LimeLite 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 LimeLite'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 what is placed in its tuple space since all operations are designed to pull data to the agent. Because data cannot be pushed to others, a collaborative type of interaction is dictated by the model. 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.

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. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
4. Cugola, G., Picco, G.: Peerware: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano (2001)
5. Gelernter, D.: Generative communication in Linda. ACM Trans. on Prog. Languages and Systems **7** (1985) 80–112
6. 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)
7. Roman, G.C., McCann, P.J., Plun, J.Y.: Mobile UNITY: reasoning and specification in mobile computing. ACM Transactions on Software Engineering and Methodology **6** (1997) 250–282
8. 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
9. 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
10. 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