

A Lightweight Coordination Middleware for Mobile Computing

Chien-Liang Fok, Gruia-Catalin Roman, and Gregory Hackmann

Department of Computer Science and Engineering
Washington University in Saint Louis
Saint Louis, Missouri 63130-4899, USA
{liang, roman, gwh2}@cse.wustl.edu
<http://www.cse.wustl.edu/mobilab>

Abstract. This paper presents Limone, a new coordination model that facilitates rapid application development over ad hoc networks consisting of logically mobile agents and physically mobile hosts. Limone assumes an agent-centric perspective on coordination by allowing each agent to define its own acquaintance policy and by limiting all agent-initiated interactions to agents that satisfy the policy. Agents that satisfy this acquaintance policy are stored in an acquaintance list, which is automatically maintained by the system. This asymmetric style of coordination allows each agent to focus only on relevant peers. Coordination activities are restricted solely to tuple spaces owned by agents in the acquaintance list. Limone tailors Linda-like primitives for mobile environments by eliminating remote blocking and complex group operations. It also provides timeouts for all distributed operations and reactions, which enable asynchronous communication with agents in the acquaintance list. Finally, Limone minimizes the granularity of atomic operations and the set of assumptions about the environment. In this paper we introduce Limone, explain its key features, and explore its capabilities as a coordination model. A universal remote control implementation using Limone provides a concrete illustration of the model and the applications it can support.

1 Introduction

Mobile devices with 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 capable of forming ad hoc networks. 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. Applications for ad hoc networks are expected to quickly grow in importance because they address challenges set forth by several important application domains.

The salient properties of ad hoc networks create many 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 have limited battery and computing power, further complicating application development. The limited functionality of mobile devices and the peer-to-peer nature of the network lead to strong mutual dependencies among devices, which have to cooperate to achieve a variety of common goals. This results in an increased need for coordination support. For example, in a planetary exploration setting, miniature rovers, each equipped with a single sensor and connected by a wireless ad hoc network, may need 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, access control, etc.) and expose too many details that complicate the programming tasks. The development and use of new programming languages is costly and entails great risks. Middleware, however, provides high-level abstractions while minimizing risk by employing the existing software infrastructure. When designed properly, middleware can divert attention from mundane concerns, like protocol development, to more fruitful areas involving application-specific goals.

Designing a coordination middleware for ad hoc networks is difficult. It must be lightweight in terms of the amount of power, memory, and bandwidth consumed. Depending on the application, it may have to operate over a wide range of devices with different capabilities: some devices, such as a laptop, may have plenty of memory and processing ability, while others, such as a sensor network Mote, may have extremely limited resources. A coordination middleware for ad hoc networks must be flexible in order to adapt to a dynamic environment; for example, in the universal remote control application, a remote held by the user must interact with a set of devices within its vicinity, which changes as the user moves. In a static sensor network, neighboring nodes may periodically enter sleep mode, run out of power, or be destroyed. Furthermore, wireless signals are prone to interference from the environment. This means the middleware must be designed to handle unpredictable message loss.

Coordination middleware facilitates application development by providing high-level constructs such as tuple spaces [1], blackboards [2], and channels [3], in place of lower-level constructs such as sockets. Tuple spaces and blackboards are both shared-memory architectures in which nodes may insert and remove data. Tuple spaces differ from blackboards in that they use pattern-matching for retrieving data; in a blackboard, the data is stored in a global database and is generally accessed by type alone. Channels are similar to sockets in that data is inserted at one end and is retrieved from the other. They differ in that the end points of a channel may be dynamically rebound.

These high-level constructs facilitate coordination by providing a layer of decoupling between nodes. In order to create a socket, the identity of the destina-

tion must be known and remains fixed. This is rather inflexible and complicates application development, particularly in ad hoc networks where connectivity is dynamic. High-level constructs, however, do not require the sender and receiver to be aware of each other. When using a tuple space or blackboard, the node that inserts data need not know the node that later extracts it. Also, since the shared space is public, multiple nodes may retrieve the same data. When using a channel, the sender need not know which node is bound to the receiving end of the channel. This level of decoupling simplifies application development because changes in connectivity no longer need to be dealt with explicitly.

This paper introduces Limone,¹ a 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 units of modularity, execution, and mobility. In a significant departure from existing coordination research, Limone emphasizes the individuality of each agent by focusing on asymmetric interactions among agents. Each agent contains an *acquaintance list* that defines a personalized view of remote agents within proximity. For each agent, Limone discovers remote agents and updates its acquaintance list according to customizable policies.

As in most coordination models, traditional Linda-like primitives over tuple spaces 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 due to packet loss or disconnection. For these reasons, Limone is resilient to message loss and unexpected disconnection. This allows Limone to function in realistic ad hoc environments in which most other models cannot.

The paper starts with a review of existing coordination models in Section 2. It then presents an overview of Limone in Section 3. Section 4 discusses the design of Limone followed by an evaluation of its performance in Section 5. The paper ends by presenting a sample universal remote control application in Section 6. Conclusions appear in Section 7.

2 Related Work

The unique characteristics of wireless ad hoc networks, such as a dynamic topology and limited node capabilities, increase the need for coordination support. To address this need, numerous coordination models have been proposed. These include JEDI [4], MARS [5], and LIME [6]. Of these, LIME is the only coordination middleware designed explicitly for ad hoc networks as it is the only model that provides a discovery protocol, which is necessary for a node to determine

¹ Limone stands for “Lightly-coordinated Mobile Network”

who it can coordinate with. We consider the other two models because they can be easily modified to function in an ad hoc network by adding a discovery protocol. In this section, we present each model and analyze their effectiveness for supporting the universal remote control application. This application consists of a universal remote held by the user that forms an ad hoc network with controllable devices within its environment. It must learn about the capabilities of a device and present a graphical interface for the user to be able to control it. Coordination support is required for the universal remote in order to discover and control the devices.

JEDI. JEDI offers a publish-subscription paradigm where nodes interact by exchanging events through a logically centralized event dispatcher. The event dispatcher may be located on a single node, or distributed across multiple nodes forming a hierarchy. An event is modelled as an ordered set of strings where the first is the name of the event, and the rest are application-specified event parameters. Nodes subscribe to events using regular expressions on the event name. When a node publishes an event, the event dispatcher passes it to all nodes subscribed to it. Since all communication is done through the event dispatcher, the publisher is decoupled from the subscriber(s).

The universal remote control application can be implemented in JEDI as follows. Devices in the environment publish *description events* that describe how they are controlled. The remote control discovers devices by subscribing to these events. Similarly, the universal remote publishes *control events* with instructions for a particular device. Each device subscribes to control events that are destined for it. The main problem with using this coordination paradigm is the lack of event persistency. When a device publishes a description event, the event dispatcher passes it to all universal remotes that have subscribed to it. But in a mobile environment, the universal remote may not be present at the time when the event was published. Thus, the device must periodically re-publish its description event to ensure all remote controls receive it. This increases bandwidth and battery power consumption.

MARS. MARS provides logically mobile agents that migrate from one node to another. Each node maintains a local tuple space that is accessed by agents residing on it. The tuple space is enhanced with reactive programming, which allows an agent to respond to actions performed on the tuple space. An agent can only coordinate with other agents that reside on the same node. Agent migration is required for inter-node communication. MARS adapts to mobile environments by allowing agents to “catch” connection events that indicate the presence of a remote node to which they may migrate.

The universal remote control application can be implemented in MARS as follows. Whenever a device’s agent detects the presence of a universal remote control agent, it spawns a new agent that migrates to the universal remote, and inserts a *device description tuple*. The universal remote control’s agent reacts to the insertion of this tuple, and thus learns about the device. A similar mechanism can be used whenever the universal remote control wishes to control the device.

This design is inefficient since it requires an agent migration for each operation, which is often more costly than simple message passing.

Lime. LIME is a coordination model that utilizes logically mobile agents running on physically mobile hosts. LIME offers a group membership paradigm where neighboring hosts form a group that share one or more logically centralized Linda-like tuple spaces that are enhanced with reactive programming. Agents coordinate by exchanging tuples through the tuple space. The agent that inserts a tuple is decoupled from the agent(s) that retrieve it. Reactive programming allows the system to notify an agent when a particular tuple is in the tuple space, which eliminates the need for polling. LIME provides strong atomicity and functional guarantees by utilizing distributed transactions. For example, when two groups merge, the merging of the two tuple spaces is done as a single atomic transaction involving all hosts. While powerful, this atomicity comes at a cost since it requires a symmetric relationship between nodes and assumes connectivity throughout the transaction, which may be difficult to guarantee in an ad hoc environment.

The universal remote control can be implemented in LIME as follows. The remote discovers the presence of controllable devices by reacting to *device description tuples* that are inserted by the devices. To control the devices, the remote inserts *control tuples* that the targeted device reacts to. The main problem with this implementation is the underlying symmetric engagement enforced by group membership. Since LIME forms groups of hosts, all the devices must be in the same group as the universal remote. Thus, each device must coordinate with all other devices when performing a distributed transaction. This presents a scalability problem.

The universal remote application is difficult to implement in existing coordination models. Some impose too much overhead while others limit flexibility by relying on strong assumptions about the network. In the next section, we introduce a new coordination model called Limone that addresses the drawbacks pointed out in this section.

3 Model Overview

Limone has been shaped by a set of highly pragmatic software engineering concerns, particularly, the desire to facilitate rapid mobile application development under realistic environmental assumptions. 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 possible and, under this minimalist assumption, it offers a precise and reasonable set of functional guarantees.

The willingness to accommodate a high degree of uncertainty about the physical state of the system raised important research questions regarding the choice

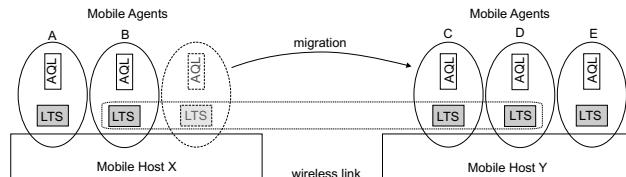


Fig. 1. An overview of Limone. Agents are represented as ovals. Each agent owns a local tuple space (LTS) and an acquaintance list (AQL). In this example, agent C is shown as migrating to host Y without a change in its acquaintance list, which consists of B and D. The dotted rectangle surrounding the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

of coordination style and associated constructs. A minimalist philosophy, combined with the goal of achieving high 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 customizable admission policies. From the application’s perspective, 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. 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 by a customizable *engagement policy*. Since each agent has different neighbors and engagement policies, the context of each agent may differ from that of its peers.

Many existing coordination models for mobility in ad hoc environments presume a symmetric and transitive coordination relation among agents that is not scalable. In such systems, every node must coordinate with every other node in the group. As the number of nodes increase, the likelihood that some nodes move out of range also increases. This results in frequent group reconfigurations, which consumes valuable resources. By allowing an agent to restrict coordination to only agents it is interested in, Limone scales better to dense ad hoc networks and

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, thus minimizing wasted memory and processor cycles. This asymmetry increases the level of decoupling among agents and results in a more robust coordination model that requires fewer assumptions about the underlying network [7].

Limone accomplishes explicit data access in a manner similar to that employed by most other coordination models. Each agent owns a single tuple space that provides operations for inserting and retrieving tuples. 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 are performed and is customizable. By default, it allows all requests 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. For example, 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 by preventing all requests that would remove it from executing. Using this read-only tuple, a certain degree of secrecy and authentication can be achieved. 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 knows that the tuple is secret if it is able to decrypt it using its private key. It also knows that the tuple was sent by the reference agent if it can decrypt it using the reference agent's public key. This example illustrates how the remote operation manager can be configured to perform just one complex task, e.g., authentication. The possibilities are endless.

The fact that Limone uses one tuple space within each agent 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 configuring the engagement policy so as to only include agents on the local host in the acquaintance list.

Reactive programming constructs enable an agent to automatically respond to the appearance of particular tuples in 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 it in its reaction registry. Once registered, Limone automatically prop-

agates the reaction to all agents in the acquaintance list that satisfy certain properties specified by the reaction (e.g., the agent’s name or location). At the receiving end, the operation manager determines whether to accept the reaction. If accepted, the reaction is placed into the reaction list, which holds the reactions that apply to the local tuple space.

When the tuple space contains a tuple satisfying 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 a 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 [8], 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 access to 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.

4 Design

Limone provides a runtime environment for agents via the Limone Server, a software layer between the agent and the underlying operating system. 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 2. 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 relevant based on their profiles. The operation manager specifies which remote operation requests are accepted. This section describes how Limone fulfills its responsibilities and is organized around the key elements of the run-time environment, i.e., agent discovery, management, reactions, and agent mobility.

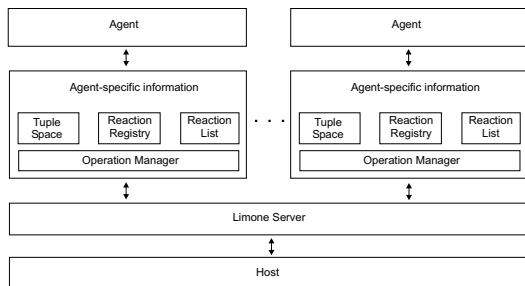


Fig. 2. The overall structure of Limone.

Discovery Mechanism. Since network connectivity between hosts in ad hoc networks is dynamic, 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 construct in Limone because it requires periodic broadcasts that consume 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 on which the agent resides 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.

When an agent receives a beacon, it passes the profiles within it 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 acquaintance list. If a particular agent's profile is already in the list, the acquaintance handler ensures that it is up to date and that it still satisfies the engagement policy.

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

The acquaintance list, shown in Figure 3, 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, and thus does not require a distributed transaction.

The removal of a remote agent's profile from the acquaintance list signifies *disengagement* between the reference agent and the remote agent. When this

<p>ABSTRACT STATE: A set of profiles, $\{p_1, p_2, \dots\}$</p> <p>INTERFACE SPECIFICATION:</p> <p>boolean contains(AgentID aID) — Returns <i>true</i> if the list contains a profile that has the specified AgentID.</p> <p>Profile[] getApplicableAgents(ProfileSelector[] pss) — Returns all of the profiles within the list that match any of the specified profile selectors.</p>

Fig. 3. Acquaintance list.

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. All application data is stored in individually owned tuple spaces. Each 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. For example, in the universal remote application, the following may represent a tuple created by the remote destined for a device:

$$\text{tuple}\{\langle\text{"type"}, \text{String}, \text{"command"}\rangle, \\ \langle\text{"device ID"}, \text{String}, \text{"CD Player"}\rangle, \\ \langle\text{"instruction"}, \text{String}, \text{"play"}\rangle\}$$

Agents use templates to access tuples 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. A constraint matches a field if the field’s name, type, and value satisfies the constraint function. For example, the following template matches the message tuple give above:

$$\text{template}\{\langle\text{"type"}, \text{String}, \text{valEq}(\text{"command"})\rangle \\ \langle\text{"device ID"}, \text{String}, \text{valEq}(\text{"CD Player"})\rangle\}^2$$

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, or its arity, to create a template for it.

² $\text{valEq}(p)$ is a constraint function that returns *true* if the value within the field is equal to p .

<p>INTERFACE SPECIFICATION:</p> <p>void out(Tuple t) — Places a tuple, <i>t</i>, into the tuple space.</p> <p>Tuple rd(Template template) — Blocks until a tuple matching the template is found within the tuple space. Returns a copy when found.</p> <p>Tuple rdp(Template template) — Returns a tuple from within the tuple space that matches the template, or ε if none is found.</p> <p>Tuple[] rdg(Template template) — Blocks until a tuple matching the template is found within the tuple space. When found, a copy of all matching tuples are returned.</p> <p>Tuple[] rdgp(Template template) — Returns all tuples from within the tuple space that match the template, or ε if none is found.</p> <p>Tuple in(Template template) — Blocks until a tuple matching the template is found within the tuple space. When found, the tuple is removed and returned.</p> <p>Tuple inp(Template template) — Removes and returns a tuple from within the tuple space that matches the template, or ε if none is found.</p> <p>Tuple[] ing(Template template) — Blocks until a tuple matching the template is found within the tuple space. When found, all matching tuples are removed and returned.</p> <p>Tuple[] ingp(Template template) — Removes and returns all tuples from within the tuple space that match the template, or ε if none is found.</p>
--

Fig. 4. Operations on the local tuple space.

Local Tuple Space Operations. The operations an agent can perform on its local tuple space are shown in Figure 4. 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 distributed transaction because they are performed locally on a single agent.

Remote Tuple Space Operations. To allow for inter-agent coordination, an agent can request a remote agent to perform an operation on their tuple space. To do this, Limone provides remote operations **out**, **inp**, **rdp**, **ingp**, and **rdgp**, as shown in Figure 5. 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 is 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 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

<p>INTERFACE SPECIFICATION:</p> <p>void out(AgentLocation loc, Tuple t) — Asks the agent located at loc to place a tuple in its tuple space.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>

Fig. 5. Operations on a remote tuple space.

a communication failure. 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 operation will time-out and return ε . To resolve the case when an operation times out while the response is still in transit, each request is enumerated, 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 an application-defined call-back function that is executed by the agent that created it when a tuple that matches the reaction's template 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 should be propagated 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 atomically. To prevent deadlock, the call-back function cannot perform blocking operations.

The list of profile selectors within the reactive pattern determines where to propagate the reactive pattern. Implementation-wise, a profile selector is a tem-

<p>ABSTRACT STATE: — A set of reactions, $\{r, \dots\}$</p> <p>INTERFACE SPECIFICATION:</p> <p>ReactionID addReaction(Reaction rxn)— Adds a reaction to the reaction registry and returns the reaction’s ReactionID.</p> <p>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.</p> <p>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.</p> <p>Reaction get(Profile profile) — Retrieves all reactions containing profiles that match the given profile or ε if no reaction matches.</p>
--

Fig. 6. Reaction Registry.

plate 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 application, a controllable device would have the following profile:

$$\text{profile}\{\{\text{"type"}, \text{String}, \text{"Device"}\}\}$$

and a reaction created by the universal remote control would contain the following profile selector to restrict its propagation to device agents:

$$\text{profile selector}\{\{\text{"type"}, \text{String}, \text{valEq1}(\text{"Device"})\}\}$$

In this case the reactive pattern will propagate to any agent whose profile contains a property called “*type*,” with a *String* value equal to “*Device*.”

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 once 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. ONCE_PER_TUPLE reactions remain registered after firing, thus firing once for each matching tuple. These 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.

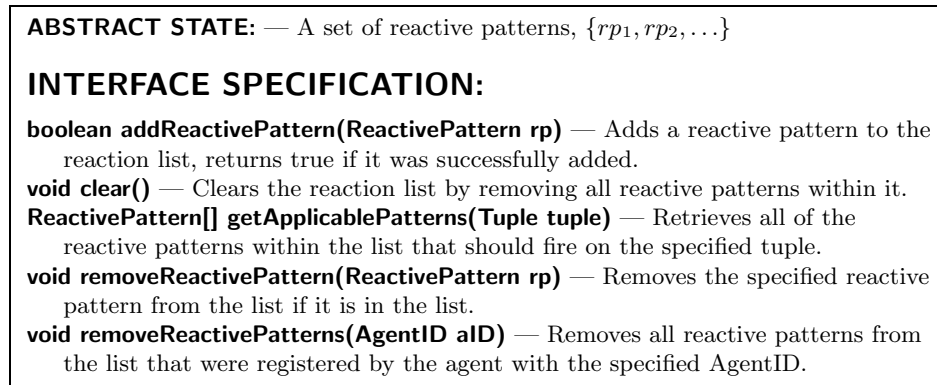


Fig. 7. Reaction List.

Two additional state components, the *reaction registry* and *reaction list*, are required for the reaction mechanism. The reaction registry, shown in Figure 6, 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 a reaction fires.

The reaction list, shown in Figure 7, 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.

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 μ Code [9]. μ Code provides primitives to support light-weight mobility preserving code and state. Of particular interest is the μ CodeServer and mobile agent. A mobile agent maintains a reference to a μ CodeServer and provides a `go(String destination)` method that moves the agent's code and data 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 μ Code by running a μ CodeServer alongside each *Limone* Server and having the *Limone* agent extend μ Agent. By extending μ Agent, 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 μ Code. Just prior to migration, the agent first deregisters all of its

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

Fig. 8. Application code size and round-trip message passing time using reactions as a trigger, averaged over 100 rounds.

reactive patterns from remote agents, and removes its profile from the beacons. Once on the new host, the agent resumes the broadcasting of its beacons. This allows remote agents to re-engage with the agent at its new location.

5 Evaluation

A prototype implementation of Limone has been developed using Java. The prototype strictly adheres to the model given in Section 3, where each construct is a distinct object that implements the interface and behavior described in Section 4.

To use Limone, a Limone Server must be created. The application can customize a variety of Limone Server attributes including the communication port and single-cast protocol. In our current implementation, the two protocols supported are TCP and UDP. By supporting either protocol, our implementation functions on small devices that cannot support the overhead of TCP. However, a Limone Server can only communicate with other Limone Servers that use the same protocol. The Limone Server listens for incoming messages and beacons. It is also for periodically broadcasting beacons, which contain the profile of all agents residing on it.

Once a Limone Server has been created, the application can load its 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.

As a testament to how lightweight Limone is, its jar file is 111.7KB. 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 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 8. The results show that Limone adds some overhead over raw sockets, but not as much as LIME. Interestingly, while Limone decreases the amount of code the application

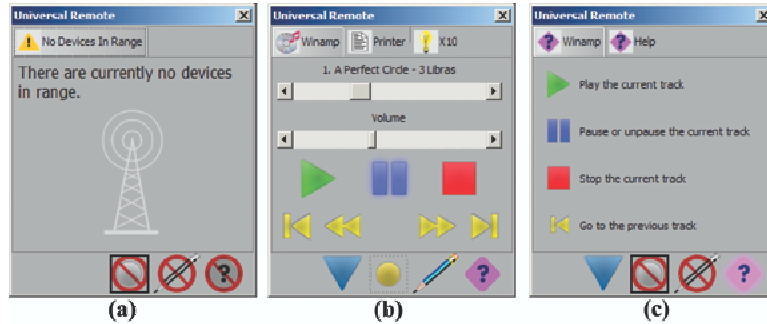


Fig. 9. The Universal Remote (a) at start-up before finding any devices, (b) after finding several available devices, and (c) displaying help for the selected device

developer must write, it still requires more code than LIME. This is due to Limone’s more expressive pattern matching mechanism and engagement policy.

6 Sample Application

This section presents the universal remote control application we developed using Limone. Limone is ideal since it automatically discovers all controllable devices within range of the remote control, allows the devices’ state to be shared amongst multiple remotes by allowing them to access the same tuples, and is lightweight enough to run on embedded devices such as electrical appliances.

When the Universal Remote is started, it briefly displays the notice shown in Figure 9(a) while it finds devices in range. As soon as it finds the first device in range, it begins displaying the devices in a tabbed list, as shown in Figure 9(b). Devices that go in or out of range are added to or removed from the list of tabs, ensuring that users cannot control devices that are no longer available. In addition to the controls for each device, a fixed row of controls is available along the bottom of the window to scroll the display, show or hide the grid, edit button placement, or show context-sensitive help (as shown in Figure 9(c)).

Each controllable device runs a Limone server and has an associated agent. These agents insert information about the device into their local tuple space: namely, the advertised list of controls (buttons, sliders, etc.); the “help text” associated with each of these controls; and the current state of each control.

In order to further simplify the creation of device agents, we implemented a `GenericDeviceAgent` class as well as a `DeviceDefinition` interface. The `GenericDeviceAgent` is a Limone agent that accepts any `DeviceDefinition` interface as a plug-in; this interface exposes information about the device (such as its advertised controls) to the `GenericDeviceAgent` as well as exposing specific reactions (i.e., pressed buttons or moved sliders) to the device. This allows the device agent to be implemented with little to no knowledge of Limone.

As an example, we simulated a remotely-controllable stereo by writing a device agent to control Winamp. This required implementing the eleven methods in `DeviceDefinition` interface in the `WinampAgent` class, which took about 250 lines of code and about an hour to write. The agent is started by starting a Limone server on the computer hosting Winamp, loading a `GenericDeviceAgent`, and instructing it to interface with the `WinampAgent` class.

The `GenericDeviceAgent` instantiates a `WinampAgent` instance and immediately calls its `getID()`, `getName()`, `getIcon()`, `getFunctions()`, `getHelpText()`, `getAdvertisement()`, and `getState()` methods in order to gather basic information about the device. (Though these functions are numerous, they are trivial to implement, since they only return fixed information like the device's name and a list of its functions.) Based on this information, the `GenericDeviceAgent` inserts the appropriate tuples into its tuple space, which the Universal Remote client can use to create its display.

When the Universal Remote client alters the state of a function (such as by toggling a button), it creates an `ActionTuple` that describes the change and inserts the tuple into the device's local tuple space. The `GenericDeviceAgent` reacts to this tuple and calls the `reactToButton()` or `reactToSlider()` method on the `WinampAgent` to alert it to the action. The `WinampAgent` then handles the change (such as by pausing the song if the pause button was toggled) and passes any change to the device's state (such as that the pause button is now lit) to the `changeState()` method on the `GenericDeviceAgent`.

The `GenericDeviceAgent` places this information into a `StateTuple` and inserts this tuple into its local tuple space. The Universal Remote client reacts to this `StateTuple` and updates its display accordingly.

Notably, aside from the SWT graphics library, no third-party libraries were needed in the implementation of the Universal Remote client, and no third-party libraries were needed for the implementation of the agents aside from libraries specific to each device (e.g., an X10 communication library for the X10 agent). Further, since Limone uses a small subset of the Java API, both the client and server could be run on a device with limited Java support, like a PocketPC. (The Universal Remote client was in fact initially designed to run on a PocketPC, but its performance in graphics-heavy applications like the remote turned out to be inadequate for the task.)

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 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 by 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.

References

1. Gelernter, D.: Generative communication in Linda. *ACM Trans. on Prog. Languages and Systems* **7** (1985) 80–112
2. Englemore, R., Morgan, T.: *Blackboard systems*. Addison-Wesley Publishing Company (1988)
3. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Technical Report 89-86 (1989)
4. 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
5. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. *Internet Computing* **4** (2000) 26–35
6. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: *Proc. of the 21st Int'l. Conf. on Distributed Computing Systems*. (2001) 524–533
7. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: *Proc. of the 10th Int'l. Symp. on Foundations of Software Engineering*. (2002)
8. McCann, P.J., Roman, G.C.: Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering* **24** (1998) 97–110
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