Electronic Notes in Theoretical Computer Science 54 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume54.html 18 pages

CoreLime: A Coordination Model for Mobile Agents

B. Carbunar, M. T. Valente, J. Vitek [1]

Computer Science Department
Purdue University
West Lafayette, USA

Abstract

Lime is a middleware communication infrastructure for mobile computation that addresses both physical mobility of devices as well as logical mobility of software components by providing a rich set of primitives for local and remote operations. The original Lime specification is surprisingly complex and tricky to implement. In this paper, we start by deconstructing the Lime model to identify its core components. In a second step we attempt to reconstruct a simpler model, which we call CoreLime, that scales better to large and rapidly changing configurations of agents and hosts.

1 Introduction

Traditional computational models are based on the assumption that devices as well as software components are deployed before being used and that once deployed configurations are static. In the emerging field of wireless computing, in which PDAs or Java-enabled phones can establish *ad hoc* network connections and application software may control its own deployment, these assumptions do not hold. Instead new computational models are needed to ease the task of developing application codes in such fluid environments.

In the last few years a number of theoretical models, from Ambients [7] to Seal [16] have been put forward as foundational models for mobile computation. Each of these models centers around some concept of migratory computation, or *mobile agent*, which is used to abstract both physical and logical mobility. In practice, dealing with mobile hosts is much more challenging than with software mobility. Thus mobile agent systems, such as Aglets [11], JavaSeal [4] and many others, mainly focus on providing a host execution

[1] E-mail: {carbunar,valente,jv}@cs.purdue.edu

environment along with support for logical mobility. Support for physical mobility is, as of this writing, mostly lacking and the communication primitives provided in these systems are left overs from traditional static computational models. The latter is arguably the largest obstacle to success of mobile agent technologies.

Designing communication mechanisms for mobile environments is a challenging task. Mobile systems have markedly different characteristics from traditional distributed or concurrent systems. Communication in a mobile system is transient and opportunistic, applications need to take advantage of available resources without assuming their continued connectivity, but considering the possibility of sudden interruptions. Communication in mobile systems is also anonymous, relying on the services being offered and not on identity of the entity providing those services.

In 1999 Murphy, Picco and Roman [15,12] introduced *Lime*, an elegant combination of Gelernter's Linda [9] with reactive programming. The design's goal being to provide a simple communication model for mobile environments. Lime introduces the notion of *transiently shared tuple spaces*. In the model each mobile entity is equipped with its own individual tuple space which moves whenever that entity moves. These individual tuple spaces are silently merged by Lime as soon as several agents are located on the same host, thus creating temporary sharing patterns that change as agents enter and leave the host. Furthermore *ad hoc* federations of hosts can be created dynamically. In this case, Lime merges the tuple spaces of each host into a single seamless federated tuple space.

This paper documents our attempts to understand Lime and to provide a scalable implementation of its key ideas. Our first contribution is a formalization of the core concepts of the model as a process calculus. This gives us well understood starting point for reasoning about Lime programs as well as a specification for implementers. Our second contribution is the definition of CoreLime, a simple and scalable basic calculus. This yields a variant of Lime without some of its potentially costly features which we intend as a basis for building next generation Lime implementations.

The structure of this paper is the following. Section 2 provides an informal introduction to Lime and its Java implementation. Section 3 formalizes the core elements of Lime as a process calculus. Section 4 discusses some of the rough spots of the specification including the strong atomicity requirements placed on federated spaces. Section 5 introduces our CoreLime model that simplifies some of the assumptions of Lime to make it easier to implement and more efficient. Finally, Section 6 relates Lime to other tuple space based communication infrastructures.

2 Lime: Middleware for Mobile Environments

This section introduces the Lime middleware communication infrastructure for mobile environments. Lime was specified informally in a 1999 paper [15] and large parts of the model were formalized in Mobile Unity notation in Murphy's thesis [13]. Furthermore, a Java implementation is available from `www.sourceforge.org`. When necessary we will differentiate between the Lime implementation (denoted $\text{Lime}_{\mathbf{imp}}$) and the Lime specification (denoted $\text{Lime}_{\mathbf{spec}}$). Examples will be written both in Java syntax and the Lime calculus of Section 3.

Lime basics.

Programs in Lime are composed of *agents* equipped with possibly many tuple spaces. Agents run on *hosts* with active tuple space managers. The basic tuple space operations available in Lime are familiar from Linda systems. Agent can deposit data in a tuple space with a non-blocking `out` operation, remove a datum with a blocking `in` or a non-blocking `inp`. They can further obtain a copy of a tuple with `rd` and `rdp`. These last operations do not modify the tuple space.

Location-aware Computing.

Lime lets agents perform operations on tuple spaces of other agents by the means of location parameters. Location parameters restrict the scope of tuple space operations. For the `out` operation, a location parameter can be used to specify the destination agent of a tuple. Its semantics is that Lime will deliver the tuple to the destination as soon as the destination agent becomes reachable. While the destination agent is not reachable tuples remain under the ownership of their creator. One way to represent this ownership information is to think of each tuple as having two additional fields `current` and `final` such that `current` denotes the current owner of the tuple and `final` its destination. A small producer/consumer example with two agents exchanging data over a common space is shown next:

$$a_h[\,!\,\mathbf{out}\,\langle x\ b\ s\rangle\,] \mid b_h[\,!\,\mathbf{in}\,\langle ?\ b\ b\ s\rangle, y\,]$$

The example is mostly straightforward. A producer agent outputs tuples consisting of a value x to the tuple space s, while the consumer agent continuously attempts to retrieve them.

Reactive programming.

On top of the standard Linda primitives, Lime introduces the concept of *reactions*. A reaction can be viewed as a triple `(t, s, p)` consisting of a tuple space `t`, a template `s` and a code fragment `p`. The semantics of a reaction is that whenever a tuple matching `s` is deposited in `t`, the code fragment `p` should be run. The main difference between the blocking `rd` and reactions is that *all*

matching reactions are guaranteed to be run when a matching tuple is found. Furthermore, Lime specifies that reactions are atomic; in other words while `p` executes, *no other* tuples space operation may be processed. Atomicity ensures that reactions always execute in a consistent state. The code of a reaction is allowed to perform tuple space operations and may thus trigger other reactions. Lime executes reactions until no more reactions are enabled. To avoid deadlocks reactions are not allowed to issue blocking tuple space operations such as `in` or `rd`. By default, reactions are fired once, but it is also possible to specify that a reaction be fired *once per tuple*. Continuing the previous example, the producer agent inserts new tuples only when receiving an acknowledgment from the receiver.

$$a_h[\,\mathbf{out}\,\langle x\ b\ s\rangle \mid \mathbf{react}_{\text{once-p-t}}\langle \texttt{"ack"}\ a\ a\ s\rangle, y.\mathbf{out}\,\langle x\ b\ s\rangle\,] \mid$$
$$b_h[\,!\mathbf{in}\,\langle ?\ b\ b\ s\rangle, x.\mathbf{out}\,\langle \texttt{"ack"}\ a\ s\rangle\,]$$

In this example the producer agent inserts the first value, and the next ones will be produced in the reaction's body when it receives the acknowledgment from the consumer.

Transiently Shared Spaces.

By default, the tuple spaces of different agents are disjoint and agents can not use tuple spaces to communicate. The key innovation in Lime is to support a flexible form of tuple space sharing referred to as *transient sharing*. An agent can declare that some of its tuple spaces are shared. The Lime infrastructure will then look for other spaces, belonging to different agents, with the same name and silently merge them into a single apparently seamless space. The sharing remains in effect as long as the agents are co-located.

Consider a simple example in which two agents have a tuple space `s`. The tuple space become transiently shared when the second agent migrates to the same host as the first.

$$a_h[\,\mathbf{out}\,\langle x\ b\ s\rangle\,] \mid b_{h'}[\,\mathbf{move}\ h.\mathbf{in}\,\langle ?\ b\ b\ s\rangle, y\,]$$

The last and most ambitious part of Lime is the support for *federated spaces*. A federated space is a transiently shared tuple space that spans several hosts. Federations arise as a result of hosts issuing the `engage` command. Hosts can leave a federation by issuing an explicit `disengage` command. The semantics of Lime operations are not affected by federations, it is up to the implementation to provide the same guarantees as in the single host case. This complicates the implementation and imposes some constraints on the use of Lime primitives. In particular, $\text{Lime}_{\mathbf{imp}}$ limits strong reactions to a single host and introduces weak reactions. A weak reaction may be scoped over multiple hosts, but it adds an asynchronous step between identification of the tuple and execution of the reaction code.

This concludes our quick overview of Lime. Interested readers are referred to [12] for a more detailed presentation. We now turn to our attempt to build

a model for the core of Lime.

3 Deconstructing Lime

This section presents a language that formalizes the coordination model proposed by Lime. We depart from Murphy's formalization by choosing an operational semantics in the style of the asynchronous π-calculus [10,2,1].

The main departure from the π-calculus is the use of generative communication operations instead of channel-based primitives. The idea of embedding a Linda-like language in a process calculus has been explored in depth in previous work [5,8].

3.1 A slice off Lime

We start with a presentation of the basic characteristics of the Lime calculus, a stripped down version of Lime. Certain features were omitted in this formalization, as our goal is to provide a convincing model of the heart of Lime rather than strive to be exhaustive.

The main differences between our model and $\text{Lime}_{\text{spec}}$ are that we omit modeling engagement and disengagement of hosts[2], instead all hosts are joined in a single large federation, and that we add an explicit notion of agents to model formally **move** operations.

Table 1 defines the syntax of the Lime calculus. We assume a set of *names* N ranged over by meta-variables, a, s, h, x. *Basic values*, ranged over by v, consist of names and tuples. *Tuples* are ordered sequences of values $\langle v_1, \ldots, v_n \rangle$. A *tuple space* T is a multiset of tuples. We use the symbol '?'$\in N$ to denote the distinguished unspecified value. As usual this value is used to broaden the scope of matching operations.

$$
\begin{array}{l}
Prog ::= A\,,\ T\,,\ X \\
A ::= \varepsilon \mid a_h[\,P\,] \mid A \\
P ::= \mathbf{0} \mid P \mid Q \mid !\,P \mid (\nu\, x)\,P \mid \mathbf{out}\, v \mid \mathbf{in}\, v, x.P \mid \mathbf{rd}\, v, x.P \mid \\
\qquad \mathbf{move}\ h.P \mid \mathbf{react}\ v, x.P
\end{array}
$$

Table 1
Lime calculus syntax

A configuration is a pair composed of a set of agents A, a tuple space T and a global set of names X. Each agent $a \in A$ is written $a_h[\,P\,]$, where P is the process running inside the agent and h the name of the host on which the

[2] Modeling engagement is not particularly difficult see [13,6] but it tends to complicate the semantics. We will have more to say about (dis)engagement in Section 4.

agent is running. In this model all agent tuple spaces are modeled by a single global tuple space T. Additional information attached to each tuple will let us distinguish ownership and current location of tuples. Similar to $\text{Lime}_{\text{spec}}$, agents can have multiple private tuple spaces. In the Lime calculus these are represented by disjoint views over the global tuple space T. These private tuple spaces are identified by names, and any two private tuple spaces with the same name are considered to be transiently shared. The names used over several hosts in the system are recorded in the set X, ensuring their unicity.

Processes are ranged by P and Q. The first four process primitives (inert process, parallel composition, replication and name creation) follow the asynchronous π-calculus. The inert process **0** has no behavior. Parallel composition of processes $P \mid Q$ denotes two processes executing in parallel. Replication of processes $!P$ denotes an unbounded number of copies of P executing in parallel. The restriction operator $(\nu\, x)\, P$ generates a fresh name x lexically scoped in process P. In our model, names are used to denote agents, hosts, tuple spaces, as well as primitive values.

Tuple spaces are accessed using Linda primitives extended with location-aware arguments. The syntax of the remaining operations is described in Table 2.

Operation	Description
out $\langle v\ a\ s \rangle$	The operation outputs v in the tuple space s of agent a. If the destination agent a is not connected, the tuple remains *misplaced* in the tuple space of the agent that requested the operation until agent a become connected.
in $\langle v\ a\ a'\ s \rangle, x$	The operation blocks until a value v' that matches v is found in tuple space s of agent a with final destination a'. If several tuples are found, one is chosen non-deterministically. Value v' is then removed from the space and bound to x.
rd $\langle v\ a\ a'\ s \rangle, x$	The operation behaves like an **in** except that the matching tuple is not removed from the tuple space.
react $\langle v\ a\ a'\ s \rangle, x.P$	The operation registers a reaction associated to the template v and whose body is process P. When a tuple v' that matches $\langle v\ a\ a'\ s \rangle$ is found in the tuple space, the reaction is triggered and process P executes atomically with x bound to v'.
move h	Moves the current agent to host h. The tuples of the agent are removed from the current host tuple space and inserted in the tuple space of host h.

Table 2
Lime calculus operations.

3.2 Semantics of Lime

We now give an operational semantics for the Lime calculus. For clarity we split the semantics in three sets of rewrite rules. The semantics is defined in Tables 3 and 4 and will be detailed next.

Primitive operations.

The first set of rewrite rules defines tuple space operations, and is of the form

$$A\,,\,T\,,\,X \quad \rightarrow \quad A'\,,\,T'\,,\,X'$$

where a configuration is a pair A, T, X such that A is a set of agents T is a tuple space, and X is a global set of names. Each step of reduction represent the effect on the program and tuple space of executing one Lime primitive operation.

The input (**in** $v, x.P$) and read (**rd** $v, x.P$) operations try to locate a tuple v' that matches v. If one is found, free occurrences of x are substituted for v' in P. In the case of the input, the tuple is removed from the space. The definition of pattern matching, written $v \leq v'$, allow for recursive tuple matching. Values match only if they are equal or if the unspecified value occurs on the left hand side.

Output (**out**) is asynchronous in Lime, and thus has no continuation. Each output tuple $\langle v\ a\ s\rangle$ is first transformed into a Lime value tuple, i.e. $\langle v\ a\ a'\ s\rangle$, and added to the global space. The Lime value tuple format has two agents names, a is the current agent that "owns" the tuple and a' is the destination agent. We say that a tuple for which $a \neq a'$ is misplaced. This can occur only if the destination is not connected[3] . The auxiliary function mkt makes a new Lime value tuple. If it can not locate the destination the tuple will be misplaced otherwise the tuple will be delivered.

Agent move operations (**move** $h.P$) change the location of the current agent. Furthermore, an auxiliary function mvt moves all the tuples to the new host.

Finally, reaction operation (**react** $v, x.P$) creates a Lime reaction tuple and deposits it in the global space. Here v is expected to have the form $\langle v'\ a'\ a''\ s\rangle$ such that v' is the value template, a' is the current agent for the tuple to match, a'' is the destination agent of the tuple to match and s is its tuple space. The rule is:

$$a_h[\,\textbf{react}\ v, x\,.\,P \mid P'\,] \mid Q\,,\,T\,,\,X \ \rightarrow\ a_h[\,P'\,] \mid Q\,,\,\langle v\ \langle a\ h\rangle\ \langle x\ P\rangle\rangle \cup T\,,\,X$$

Reaction tuples will have the form $\langle\langle v'\ a'\ a''\ s\rangle\ \langle a\ h\rangle\ \langle x\ P\rangle\rangle$ where a is the agent that registered the reaction, h is its location, and P is the reaction's body.

[3] As we are not modeling engagement and disengagement, a tuple remains misplaced only until the destination agent is created. Adding engagement rules should not affect the semantics.

Reactions.

The second set of three rewrite rules defines the semantics of reactions. In the Lime calculus, reactions are stored in the tuple space, as distinguished tuples hidden from normal user code. Thus to evaluate a reaction, we need only have a tuple space as it contains both normal data and the reactions defined over that data. The rules are of the form

$$T \quad \rightsquigarrow_S \quad T'$$

where T is a tuple space and S is the multiset of tuples that are candidates to trigger a reaction. All candidates in S will be examined. When all reactions have completed executing, the new tuple space T' is returned. In the simplest case, if there are no candidates the global tuple space is left as is:

$$\frac{}{T \rightsquigarrow_{\{\}} T}$$

If there is a candidate tuple, but it does not trigger any reaction, the rules discard it and proceed to analyze the remaining candidates:

$$\frac{T' \rightsquigarrow_S T''}{T \rightsquigarrow_{v \cup S} T''} \qquad \text{if } \nexists \langle v' \ \langle a \ h \rangle \ \langle x \ P \rangle \rangle \in T \text{ s.t. } v \leq v'$$

Finally, if a reaction matching one of the candidates has been found, then the reaction is removed from the global tuple space. A dummy agent is created under a fresh name to run the reaction's body. Once the process terminates, the resulting tuple space T'' is used to recursively look for other reactions that are ready to fire. We assume that move commands may not occur in the body of the reaction. The rule is:

$$\frac{(\nu \, r) \, r_h[\, P\{v'/x\} \,] \, , \, T' \, , \, X \ \stackrel{*}{\Rightarrow} (\nu \, r) \, r_h[\, \mathbf{0} \,] \, , \, T'' \, , \, X' \quad T'' \rightsquigarrow_{v \cup S} T'''}{T \rightsquigarrow_{v \cup S} T'''}$$

$$\text{if } \ T = \langle v' \ \langle a \ h \rangle \ \langle x \ P \rangle \rangle \cup T' \ \wedge v \leq v'$$

Note that the candidate tuple is kept for the recursive step as there may be several reactions matching a single tuple.

Global computation.

The last set of two rewrite rules simply combines the primitive rules with the reaction rules and specifies that after every primitive step, a step of reaction is run. The rules are of the form

$$A \, , \, T \, , \, X \quad \Rightarrow \quad A' \, , \, T' \, , \, X'$$

where a configuration is a pair A, T, X such that A is a set of agents, T is a tuple space and X is a global set of names. The main rule used to trigger reactions after each primitive step is:

$$\frac{A \, , \, T \, , \, X \ \rightarrow \ A' \, , \, T' \, , \, X \qquad T' \rightsquigarrow_S T'' \quad S = T' - T}{A \, , \, T \, , \, X \ \Rightarrow \ A' \, , \, T'' \, , \, X}$$

The set of tuples S that are considered as candidates for triggering reactions is obtained as the difference between the state of the global tuple space before the primitive operation and after the primitive operation $S = T' - T$.

Reductions

$\underline{\rightarrow}$:

$$a_h[\,\mathbf{in}\,v, x.P \mid P'\,] \mid Q\,,\, v' \cup T\,,\, X \;\rightarrow\; a_h[\,P\{v'/x\} \mid P'\,] \mid Q\,,\, T\,,\, X \tag{T1}$$

$$a_h[\,\mathbf{rd}\,v, x.P \mid P'\,] \mid Q\,,\, v' \cup T\,,\, X \;\rightarrow\; a_h[\,P\{v'/x\} \mid P'\,] \mid Q\,,\, v' \cup T\,,\, X \tag{T2}$$

$$a_h[\,\mathbf{out}\;v' \mid P\,] \mid Q\,,\, T\,,\, X \;\rightarrow\; a_h[\,P\,] \mid Q\,,\, v \cup T\,,\, X \tag{T3}$$

$$a_h[\,\mathbf{move}\;h'\,.\,P \mid P'\,] \mid Q\,,\, T\,,\, X \;\rightarrow\; a_{h'}[\,P \mid P'\,] \mid Q\,,\, T'\,,\, X \tag{T4}$$

$$a_h[\,\mathbf{react}\;v, x\,.\,P \mid P'\,] \mid Q\,,\, T\,,\, X \;\rightarrow\; a_h[\,P'\,] \mid Q\,,\, \langle v\;\langle a\;h\rangle\;\langle x\;P\rangle\rangle \cup T\,,\, X \tag{T5}$$

$\underline{\rightsquigarrow}$:

$$\frac{(\nu\, r)\, r_h[\,P\{v'/x\}\,]\,,\, T'\,,\, X \;\stackrel{*}{\Rightarrow}\; (\nu\, r)\, r_h[\,\mathbf{0}\,]\,,\, T''\,,\, X \quad T'' \rightsquigarrow_{v \cup S} T'''}{T \rightsquigarrow_{v \cup S} T'''} \tag{R1}$$

$$\frac{T' \rightsquigarrow_S T''}{T \rightsquigarrow_{v \cup S} T''} \tag{R2}$$

$$\frac{}{T \rightsquigarrow_{\{\}} T} \tag{R3}$$

$\underline{\Rightarrow}$:

$$\frac{A\,,\, T\,,\, X \;\rightarrow\; A'\,,\, T'\,,\, X \qquad T' \rightsquigarrow_S T'' \quad S = T' - T}{A\,,\, T\,,\, X \;\Rightarrow\; A'\,,\, T''\,,\, X} \tag{G1}$$

$$\frac{A\,,\, T\,,\, X \equiv A'\,,\, T\,,\, X' \qquad A'\,,\, T\,,\, X' \;\Rightarrow\; A''\,,\, T'\,,\, X'}{A\,,\, T\,,\, X \;\Rightarrow\; A''\,,\, T'\,,\, X'} \tag{G2}$$

The rules are subjected to the following side conditions:

(T1) if $v \leq v'$

(T2) if $v \leq v'$

(T3) $v = mkt(v', a, h, Q)$

(T4) $T' = mvt(a, h', T)$

(R1) if $T = \langle v'\;\langle a\;h\rangle\;\langle x\;P\rangle\rangle \cup T' \;\wedge v \leq v'$

(R2) if $\nexists \langle v'\;\langle a\;h\rangle\;P\rangle \in T$ s.t. $v \leq v'$

Table 3
Lime calculus operational semantics

4 Critical Assessment of Lime

During our evaluation we found several inefficiencies in both $\text{Lime}_{\mathbf{spec}}$ and $\text{Lime}_{\mathbf{imp}}$ which we believe must be addressed if Lime is to gain widespread acceptance. These problems stem from the strong atomicity and consistency imposed by $\text{Lime}_{\mathbf{spec}}$. Even when weakened in the implementation those requirements make Lime implementations overly complex, full of potential synchronization problems and quite inefficient. Even worse from a user point of view, the cost of the advanced features is paid even by applications that do not use them.

Thus, we proceed listing some rough spots of both $\text{Lime}_{\mathbf{spec}}$ and $\text{Lime}_{\mathbf{imp}}$.

Structural Congruence Rules

$P \mid Q \equiv Q \mid P$ (SC1)

$! P \equiv P \mid ! P$ (SC2)

$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$ (SC3)

$P \mid \mathbf{0} \equiv P$ (SC4)

$(\nu\, x)\,(\nu\, y)\, P \equiv (\nu\, y)\,(\nu\, x)\, P$ (SC5)

$P \equiv Q \Rightarrow (\nu\, x)\, P \equiv (\nu\, x)\, Q$ (SC6)

$(\nu\, x)\,(P \mid Q) \equiv P \mid (\nu\, x)\, Q,$ if $x \notin fn(Q)$ (SC7)

$P \equiv Q \Rightarrow a_h[\,P\,]\,,\, T\,,\, X \equiv a_h[\,Q\,]\,,\, T\,,\, X$ (SC8)

$(\nu\, x)\, a[P] \equiv a[(\nu\, x)\, P],$ if $x \neq a$ (SC9)

$(\nu\, x)\,(a_h[\,P\,] \mid b_{h'}[\,Q\,]) \equiv (\nu\, x)\, a_h[\,P\,] \mid b_{h'}[\,Q\,],$ if $x \neq b,\ x \notin fn(Q)$ (SC10)

$(\nu\, x)\, a_h[\,P\,]\,,\, T\,,\, X \equiv a_h[\,P\,]\,,\, T\,,\, x \cup X,$ if $x \notin X$ (SC11)

Pattern Matching Rules

$x \leq x \qquad ? \leq x \qquad \dfrac{v_1 \leq v_1' \ \ldots\ v_n \leq v_n'}{\langle v_1 \ \ldots\ v_n \rangle \leq \langle v_1' \ \ldots\ v_n' \rangle}$

Functions

$mkt(\langle v\ a'\ s \rangle, a, h, Q) = \langle v\ a'\ a'\ s \rangle,$ if $Q \equiv {a'}_{h'}[\,P\,] \mid Q'$

$mkt(\langle v\ a'\ s \rangle, a, h, Q) = \langle v\ a\ a'\ s \rangle,$ otherwise

$mvt(a, h, \{\}) = \{\}$

$mvt(a, h, \langle v\ \langle a\ h' \rangle\ \langle x\ P \rangle \rangle \cup T) = \langle v\ \langle a\ h \rangle\ \langle x\ P \rangle \rangle \cup mvt(a, h, T)$

$mvt(a, h, v \cup T) = v \cup mvt(a, h, T)$

Table 4
Structure congruence, pattern matching and auxiliary functions

Readers should bear in mind that we are not trying to criticize one prototype implementation of Lime rather we are trying to find characteristics that are inherent to the model.

4.1 Reaction Livelocks

$\text{Lime}_{\text{spec}}$ requires that reactions be executed atomically until a fixed point is reached. All other tuple space operations on the current host are blocked until reactions terminate. This is heavy price to pay in a highly concurrent setting. Reaction atomicity implies that the runtime cost of a Lime **out** is entirely unpredictable.

There is another problem. Since reaction bodies are normal programs,

termination can not be guaranteed. Consider the following expression:

$$\mathbf{react}\, v, x.(!\,\mathbf{out}\, v')$$

In the Lime calculus semantics this reaction will never terminate as we require that the reaction body reduces to **0**. Thus if it ever gets triggered the entire system will be stuck. In $\text{Lime}_{\mathbf{imp}}$ similar issues arise because of the use of unrestricted Java code fragments in reaction bodies.

There is a related problem which occurs with the once-per-tuple reactions. A once-per-tuple reaction can trigger itself recursively by outputting the very tuple it is interested in. For instance consider the following program

$$\mathbf{react}_{\text{ once-p-t}} \langle v\ a\ a\ s\rangle, x.\mathbf{out}\, \langle v\ a\ s\rangle$$

This program registers a reaction interested in some value v which, whenever one such tuple is inserted in the tuple space, proceeds to output a new tuple with the same value, thus triggering itself recursively. While one may argue that this particular example can be prevented by careful coding, it is much harder to prevent independently developed applications from creating mutually recursive patterns by accident.

Non-terminating reactions present a serious problem for $\text{Lime}_{\mathbf{imp}}$. Firstly, they block the entire tuple space of the current host, and since disengagement is global and atomic, they can prevent disengagement procedures from terminating, thus blocking the entire federation.

4.2 Implementation of once-per-tuple reactions

The semantics of once-per-tuple reactions is that every tuple should be distinguishable from all others so that Lime can ensure that reactions are indeed only triggered once per tuple. In a traditional tuple space, it would be rather easy to implement such semantics by making sure that reaction are run once when tuples are inserted in the space. Lime has the additional problem that agents can move taking their tuples with them. The question then becomes: if an agent leaves a host and then comes back, are its tuples going to trigger reactions [6]. $\text{Lime}_{\mathbf{spec}}$ provides an answer to this question since it requires that every tuple be equipped with a globally unique identifier (GUID). These GUIDs solve exactly that problem.

The obvious implementation strategy for once-per-tuple reactions is then to store the GUIDs of the tuples it has already reacted to. One drawback of this approach is that reaction may need to store an unbounded amount of data to remember all tuples seen, especially if GUIDs are made sufficiently large to provide some reasonable likelihood of unicity.

In practice unicity of GUIDs can be difficult to ensure. In $\text{Lime}_{\mathbf{imp}}$ for instance, agents are moved with Java serialization. In this form it is easy to create a copy of an agent along with all of its tuples. To provide real unicity guarantees the implementation would have to protect itself against replay attacks which would complicate considerably the mobility protocols.

4.3 Federated space operations

Federated spaces are distributed data structures which can be accessed concurrently from many different hosts. $\text{Lime}_{\text{spec}}$ places strong consistency requirements on federated spaces. The challenge is therefore to find implementation strategies that decrease the amount of global synchronization required.

The approach chosen by Lime_{imp} is to keep a single copy of every tuple on the same host as it's owner agent. Federated input request are implemented by multicast over the federation. Blocking requests are implemented by weak reactions which register a strong (local) reaction on every host of the federation and a special reaction on the host of the agent that issued the input request. Then whenever one of the local reactions finds a matching tuple the originating host is notified and if the agent is still waiting for input the tuple is forwarded.

The problem with this approach is one of scalability. For every federated input operation, all hosts in the federation have to be contacted, new reactions created and registered. Then once a tuple is found, the reactions have to be disabled. From a practical standpoint having additional reactions on a host slows down every local tuple space operation as the reactions have to be searched for each output operation.

We argue that federated operations are inherently non-scalable and furthermore that they impact on the performance of applications that do not use them, even purely local applications that do not have to go to the network.

4.4 Atomicity of engagements

In Lime_{imp} hosts joining a federation must be brought to a consistent state. This boils down to making sure that all of the weak reactions that hold over the federation be enforced for the new host. For each weak reaction, a strong reaction must be registered on the incoming host. The current engagement procedure is atomic which is awkward as it means that new hosts must be serialized and that other tuple operations are blocked while they are being added to the configuration.

4.5 Atomicity of disengagements

When a host desires to leave the federation it must execute a disengage operation which atomically de-registers all weak reactions registered by agents currently on that host from all other hosts in the federation. This is a costly operation as there may be many strong reactions to disable on the hosts that make up the federation, and since it involves a global lock on the federated space.

Furthermore, one may question the choice of requiring explicit disengagement notification in the context of mobile devices. If a mobile device moves out of range or loses connectivity, it is not likely that it will have the time to send a message beforehand.

4.6 Atomicity of moves

The semantics of the Lime calculus specifies that moves are atomic. There is no clear statement about moves in $\text{Lime}_{\text{spec}}$. Making moves atomic has pleasant properties, for instance we are guaranteed that in the following configuration the non-blocking **inp** will succeed.

$$a_h[\,\mathbf{move}\,h'.\mathbf{0}\,] \;\mid\; b_h[\,\mathbf{inp}\,\langle v\ a\ a\ s\rangle, x.P\,] \quad , \quad \{\langle v\ a\ a\ s\rangle\}$$

because regardless of scheduling, the **inp** will always be run in an environment where agent a is connected, either from host h or host h'. In practice, this is of course not the case as there will be some time when a is in transit between hosts. Thus, in Lime implementations, the **inp** in the above program may not succeed. A simple way to model this behavior is translate every move into a two-step operation, the agent first moves to a distinguished host which is disconnected, in the Lime sense, from the every other host and then, in a second step, moves to its destination.

4.7 Atomicity of input operations

In Lime all input operations are atomic, even the remote ones. The presence of mobility complicates the implementation of remote input operations as the agent may try to move while waiting for a reply. The question is then what should a Lime implementation do in a configuration such as

$$a_h[\,\mathbf{move}\,h.\mathbf{0} \;\mid\; \mathbf{in}\,\langle v\ b\ b\ s\rangle, x.P\,]$$

where agent b is assumed to be remote. If the input operation is selected first, should the implementation wait for the input to complete before allowing the move. Since this is a blocking in, the wait time is unbounded. On the other hand if it allows the agent to move then it must be ready to handle the additional complexity of messages sent from b's host while a is in transit.

4.8 Summary

The semantics of Lime places very strong atomicity requirements on implementations of the model. These requirements are hard to implement in a distributed setting, and even harder when devices as well as programs are allowed to move. The next section presents a simpler model of Lime that we propose as a basis for building more robust Lime implementations.

5 Back to Basics: CoreLime

The initial goal of our research was to add security primitives to Lime, but the problems that we detected while trying to understand its implementation convinced us that we had to simplify the model. Our approach is twofold, first we will provide a simpler incarnation of Lime that we call CoreLime which is a non-distributed variant of Lime with agent mobility. The syntax

and semantics of most Lime operations is retained, the main restriction is that operations are scoped over the local host only. The second part of our research will be to define semantics for the remote operations provided in Lime. For these we plan to give a translation to CoreLime using agent mobility to specify remote effects. In this paper, we present CoreLime and hint at the translation.

5.1 Semantics of CoreLime

The main difference between Lime and CoreLime is that we tried to lift all global synchronization requirements. To do so we have restricted all operations to their local variant and rely on agent mobility as the single mechanism for modeling remote actions.

A further change to Lime is that we removed the atomicity requirement on reactions. In our variant, reactions execute concurrently to user code. This allows for a much simpler semantics without the need for auxiliary reductions. The semantics is summarized in Table 5.

The main changes required are the following. Input operations must check the location of tuples matched, any tuple retrieved by an **in** or **rd** must belong to a co-located agent. This constraint is enforced by the side condition on the transitions:

$$a_h[\,\mathbf{in}\,v, x.P \mid P'\,] \mid Q\,,\, v' \cup T\,,\, X \;\rightarrow\; a_h[\,P\{v'/x\} \mid P'\,] \mid Q\,,\, T\,,\, X$$

$$\text{if } v \leq v' \;\wedge\; loc(v') = h$$

The auxiliary function *loc* returns the host where a tuple or an agent is located. Output and move reduction can trigger reactions, these are represented by a new process R running in parallel:

$$a_h[\,\mathbf{out}\;v' \mid P\,] \mid Q\,,\, T\,,\, X \;\rightarrow\; a_h[\,P\,] \mid Q \mid R\,,\, v \cup T\,,\, X$$

$$\text{where } v = mkt(v', a, h), R = react(\{v\}, T)$$

The auxiliary function *react* will create a single new agent on the current host with as body the parallel composition of matching reactions, e.g.

$$(\nu\, r)\, r_h[\,P_1 \mid \ldots \mid P_n\,]$$

This is done for each matching tuple v such that v is substituted for the parameter of the reaction body.

Remote operations.

Removing remote operations from the semantics does not prevent an agent from accessing tuple spaces of remote hosts. To exemplify this, we show how an agent can dispatch a new agent to another host to perform a remote **in** .

$$a_h[\,\mathbf{rin}\;h', \langle v', a', a'', s\rangle, x.P \mid P'\,] \triangleq$$

$$(\nu\, y)\,(\nu\, r)\;\; a_h[\,\mathbf{in}\,\langle\langle y, ?\rangle, a, a, s\rangle, x.P \mid P'\,] \mid$$

$$r_h[\,\mathbf{move}\;h'.\mathbf{in}\,\langle v', a', a'', s\rangle, x.\mathbf{move}\;h.\mathbf{out}\,\langle\langle y, x\rangle, a, s\rangle\,]$$

Reductions

$$a_h[\,\mathbf{in}\,v, x.P \mid P'\,] \mid Q\,,\; v' \cup T\,,\; X \;\rightarrow\; a_h[\,P\{v'/x\} \mid P'\,] \mid Q\,,\; T\,,\; X \quad \text{(T1)}$$

$$a_h[\,\mathbf{rd}\,v, x.P \mid P'\,] \mid Q\,,\; v' \cup T\,,\; X \;\rightarrow\; a_h[\,P\{v'/x\} \mid P'\,] \mid Q\,,\; v' \cup T\,,\; X \quad \text{(T2)}$$

$$a_h[\,\mathbf{out}\;v' \mid P\,] \mid Q\,,\; T\,,\; X \;\rightarrow\; a_h[\,P\,] \mid Q \mid R\,,\; v \cup T\,,\; X \quad \text{(T3)}$$

$$a_h[\,\mathbf{move}\;h'\,.\,P \mid P'\,] \mid Q\,,\; T\,,\; X \;\rightarrow\; a_{h'}[\,P \mid P'\,] \mid Q \mid R\,,\; T'\,,\; X \quad \text{(T4)}$$

$$a_h[\,\mathbf{react}\;v, x\,.\,P \mid P'\,] \mid Q\,,\; T\,,\; X \;\rightarrow\; a_h[\,P'\,] \mid Q\,,\; \langle v\;\langle a\;h\rangle\;\langle x\;P\rangle\rangle \cup T\,,\; X \quad \text{(T5)}$$

The rules are subjected to the following side conditions:
(T1) if $v \leq v' \;\wedge\; loc(v') = h$
(T2) if $v \leq v' \;\wedge\; loc(v') = h$
(T3) $v = mkt(v', a, h), R = react(\{v\}, h, T)$
(T4) $T' = mvt(a, h', T), R = react(sel(a, T, T'), h', T')$

Functions

$$mkt(\langle v\;a'\;s\rangle, a, h) = \langle v\;a'\;a'\;s\rangle, \text{ if } loc(a') = h$$
$$mkt(\langle v\;a'\;s\rangle, a, h) = \langle v\;a\;a'\;s\rangle, \text{ otherwise}$$

$$react(\{\}, h, T) = \mathbf{0}$$
$$react(v \cup V, h, T) = (\nu\, r)\, r_h[\,selr(v, h, T)\,] \;\mid\; react(V, h, T)$$

$$selr(v, h, \{\}) = \mathbf{0}$$
$$selr(v, h, \langle v'\,\langle a\,h'\rangle\;\langle x'\;P\rangle\rangle \cup T) = P\{v/x'\} \mid selr(v, h, T), \text{ if } v' \leq v \wedge loc(v') = h$$
$$selr(v, h, v' \cup T) = selr(v, h, T)$$

$$mvt(a, h, \{\}) = \{\}$$
$$mvt(a, h, \langle v\;a\;a'\;s\rangle \cup T) = \langle v\;a'\;a'\;s\rangle \cup mvt(a, h, T), \text{ if } loc(a') = h$$
$$mvt(a, h, \langle v\;a'\;a\;s\rangle \cup T) = \langle v\;a\;a\;s\rangle \cup mvt(a, h, T), \text{ if } loc(a') = h$$
$$mvt(a, h, \langle v\;\langle a\;h'\rangle\;\langle x\;P\rangle\rangle \cup T) = \langle v\;\langle a\;h\rangle\;\langle x\;P\rangle\rangle \cup mvt(a, h, T)$$
$$mvt(a, h, v \cup T) = v \cup mvt(a, h, T)$$

$$sel(a, T, T') = \{v \in T' \mid (\langle ?\;a\;?\;?\rangle \leq v) \;\vee\; (\langle v'\;a'\;a'\;s\rangle \leq v \;\wedge\; \langle v'\;a\;a'\;s\rangle \in T)\;\}$$

Table 5
Semantics of CoreLime.

In this operation, an agent r is dispatched to host h', where the requested **in** is executed. When a matching tuple is found, the agent r returns to the issuing host h and outputs the value found with a tag y that identifies the operation. This value is then removed by agent a.

5.2 Implementation of CoreLime

We are currently working on an implementation of CoreLime in Java. The interface of Lime_{imp} will be mostly retained with some additional constraints to enforce the local semantics of CoreLime. We are using the SecOS implementation of [3] as an underlying tuple space engine.

6 Related Work

The operational semantics presented in this paper resemble the ambient calculus of Cardelli and Gordon [7]. However, in the ambient calculus the primitive used for reading messages is not based on pattern matching and thus communicating processes must know each other's identity. Moreover, processes cannot transparently read messages located in sibling ambients.

Busi and Zavattaro have also proposed a formalization of transiently shared tuple spaces [6]. However, they do not model reactions and do not consider the impacts of supporting federating tuple spaces in a real system. TuCSoN [14] is a coordination model intended to be associated with existing agent systems. Every host provides tuple spaces that can be used by local agents for interagent communication and to access local resources. Linda like operations can be performed remotely on them by specifying their name and the name of the host.

7 Conclusions

In this paper we proposed a formalization for Lime, a communication model for mobile systems, that addresses logical mobility as well as physical mobility of devices. We have detected potential problems in the specification and the implementation of Lime, and have introduced a smaller version of the model, called CoreLime.

As future work we plan to add security mechanisms to CoreLime. Since in a mobile environment communication is anonymous, entities do not trust each other. In a real setting there are situations where features like secure channels, the possibility to authenticate communicating parties and to restrict access to resources are desirable. We will add capabilities to control access and security filters to authenticate and restrict rights. Capabilities will provide granularity at the level of tuple spaces, by encapsulating secret tuple space names. Legitimate users will not be given the right to distribute them, and the addition of rights will be forbidden. Security filters will be wrappers around

tuple spaces, acting on them much like reactions, but reacting on all actions performed. Our future work also includes an implementation for CoreLime, and the definition of semantics for the remote operations provided in Lime.

Acknowledgments

This research was supported by a grant from Motorola, CERIAS and CAPES. The authors wish to thank the authors of Lime for open sourcing their implementation, and Amy Murphy for answering our questions.

References

[1] R. M. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. In U. Montanari and V. Sassone, editors, *CONCUR '96*, volume 1119 of *LNCS*, pages 147–162. Springer-Verlag, Berlin, 1996.

[2] G. Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[3] C. Bryce, M. Oriol, and J. Vitek. A Coordination Model for Agents Based on Secure Spaces. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 4–20, Amsterdam, Netherland, 1999. Springer-Verlag, Berlin.

[4] C. Bryce and J. Vitek. The JavaSeal Mobile Agent Kernel. In D. Milojevic, editor, *Proceedings of the 1st International Symposium on Agent Systems and Applications, Third International Symposium on Mobile Agents (ASAMA'99)*, pages 176–189, Palm Springs, May 9–13, 1999. ACM Press.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, Feb. 1998.

[6] N. Busi and G. Zavattaro. Some Thoughts on Transiently Shared Tuple Spaces. In *Workshop on Software Engineering and Mobility. Co-located with International Conference on Software Engineering*, May 2001.

[7] L. Cardelli and A. Gordon. Mobile Ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[8] R. DeNicola and R. Pugliese. A Process Algebra based on Linda. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, Berlin, 1996.

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

[10] K. Honda and M. Tokoro. On Asynchronous Communication Semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing. LNCS 612*, pages 21–51, 1992.

[11] D. B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web Journal*, 1998.

[12] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001. to appear.

[13] A. T. Murphy. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*. PhD thesis, Washigton University, St. Louis, August 2000.

[14] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190. ACM, 28 Feb. – 2 Mar. 1999.

[15] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, May 1999.

[16] J. Vitek. *The Seal model of Mobile Computations*. PhD thesis, University of Geneva, 1999.