Lime Revisited

Bogdan Carbunar, Marco Tulio Valente, Jan Vitek

Dept. of Computer Sciences
Purdue University

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 through a rich set of primitives for local and remote operations. The key innovation of Lime is the concept of *transiently shared tuple spaces*. Mobile programs are equipped with tuple spaces that move whenever the program moves, Lime support transparent sharing of tuple spaces of co-located programs. The Lime specification is surprisingly complex and tricky to implement. In this paper, we start by deconstructing the Lime model to identify its core components, then we attempt to reconstruct a simpler model, which we call CoreLime, that supports fine-grained access control and scales better to large and rapidly changing configurations of agents and hosts.

1 Introduction

Traditional computational models are based on the assumption that software and devices are deployed before used and, once deployed, that configurations are relatively static. Wireless computing and *ad-hoc* networks invalidate such assumptions as both devices and the software that runs on them may be mobile. To address this issue a number of theoretical models such as Ambients [6] and Seal [15] have adopted migratory computations as their key linguistic abstraction. In these theoretical studies migratory computations or mobile agents can model both physical and logical mobility. But in practice dealing with physical mobility has proved much more challenging than software mobility. Not surprisingly, most practical mobile agent systems [10, 3] have focused on providing mechanisms for moving code and have mostly ignored mobility of devices.

In particular, our experience with the implementation of a medium-sized agent application [11] suggest that high-level communication primitives are the main shortcoming of many agent systems. Designing communication middleware 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:** Communication patterns are shaped by the nature of an environment in which hosts are intermittently connected to the network and agents can leave a host at any time. Communication thus tends to be opportunistic; applications take advantage of resources that

happen to be available at particular time without relying on their continued availability. Communication protocols must accommodate long latencies and time outs caused by the sudden departure of an interlocutor or disconnection of the agent itself.

- **Anonymous and Untrusted:** Interactions can be based on services offered rather than on identity of the entity providing those services. Agents do not necessarily have to know each others names and locations to interact as long as the needed services are being provided. The corollary of anonymity is that interlocutors do not necessarily trust each other which implies that the communication infrastructure must provide the mechanisms needed to implement secure communication protocols.

In 1999 Murphy, Picco and Roman [14, 12] introduced *Lime*, an elegant combination of Gelernter's Linda [8] 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. Transient sharing solves several problems of tuple space communication models in the context of mobile environments. At the local level, it introduces a notion of ownership for tuples that is beneficial for resource accounting purposes. Furthermore, tuple space migration allows mobile entities to suspend an ongoing interaction and resume it whenever both agents happen to be co-located again. At the federated level, transient sharing provides a model of a distributed space in the face of mobility.

The original goal of this work was simply to extend Lime with the access control mechanisms needed to implement secure interaction between untrusted parties. Along the way we realized that the Lime specification was somewhat complex and difficult to implement and that the model appeared to have some ingrained inefficiencies. These suspicions were confirmed by preliminary experiments with a prototype implementation. This paper documents our attempts to understand Lime and to provide a scalable and secure 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, and an extension of this simple model with access control primitives. 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.

2 Lime: Middleware for Mobile Environments

This section introduces the Lime middleware communication infrastructure for mobile environments. Lime was specified informally in [14] and parts of the model were formalized in Mobile Unity [13].

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`.

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. 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`.

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 tuple for which `current` $\neq$ `final` is in transit (also called *misplaced* in Lime). Lime implementations need not maintain these fields explicitly, they are useful for the exposition though.

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.

Although the model does not provide for agent mobility, the underlying assumption is that agents can leave a host at any time. When this occurs, Lime will break up the tuple space and extract all tuples which have the departing agent in their `current` field. Transient sharing simplifies the coding of application communication protocols as explicit location parameters can be omitted to search the entire shared space.

Federated Spaces and Mobile Hosts. 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, $\text{Lime}_{\mathbf{imp}}$ introduces weak reactions and limit (strong) reactions to a single host. 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. Tuples that may trigger weak reactions are first set aside, and then the user reactions are executed atomically. In $\text{Lime}_{\mathbf{imp}}$, a weak reaction is implemented by registering one strong reaction on every node of the federation.

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 [9, 2, 1]. The main reason for the departure is that it allows for a self-contained semantics, which does not have to rely on extraneous definition. Furthermore, we hope to obtain a compact and simple formalization. The main difference between our formalism and 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 [4, 7].

We start with a presentation of the basic characteristics of the Lime calculus, a stripped down version of Lime. 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.

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 agent is running. Agent tuple spaces are modeled by a single global tuple space T. Additional information attached to each tuple will let us distinguish

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

Table 1. Lime calculus syntax

ownership and current location of tuples. Similar to $\text{Lime}_{\mathbf{spec}}$, 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 inert process $\mathbf{0}$ 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.

The **out** operation expects a tuple $v = \langle v'\ a\ s \rangle$ as argument. The first element of the tuple is the value v' (a tuple itself) to be output, a is the destination agent and s is the tuple space. The **in** and **rd** primitives expect an argument tuple $\langle v\ a\ a'\ s \rangle$ and the name of variable that will be bound to the result[1]. The argument tuple consists of the template to match v, the current owner of the desired tuple a, the destination of the desired tuple a' and the tuple space s. The unspecified value can be used to broaden the scope of input operations, e.g. if both current and destination fields are left unspecified, the entire space will be searched.

The **move** primitive that can be used by agents to migrate between connected hosts. When an agent performs this operation all the tuples that have the agent as the current location are removed from the source host, moved with it, and inserted in the destination host. he primitive **react** v, x, P is used to register a reaction on the host where the agent that executes it resides. The first argument is the tuple that has to be matched in order for the reaction to be triggered, and the second argument is a variable and the third argument a process, called the body of the reaction, that will be executed atomically upon occurrence of such an event.

[1] The $\text{Lime}_{\mathbf{imp}}$ equivalent of **out** $\langle v\ a\ s \rangle$ and **in** $\langle v\ a\ a'\ s \rangle, x$ are, respectively, `s.out(a, v)` and `x = s.in(a, a', v)`.

3.1 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 Table 2 and will be detailed next.

Primitive operations. The first set of rewrite rules defines tuple space operations, and is of the form $A\,,\,T\,,\,X \rightarrow A'\,,\,T'\,,\,X'$ 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 ($\mathbf{in}\, v, x.P$) and read ($\mathbf{rd}\, v, x.P$) 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. 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 ($\mathbf{move}\, h.P$) change the location of the current agent. Furthermore, an auxiliary function mvt moves all the tuples to the new host. Finally, reaction operation ($\mathbf{react}\, v, x.P$) 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. 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.

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 $T \rightsquigarrow_S T'$ 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. If there is a candidate tuple, but it does not trigger any reaction, the rules discard it and proceed to analyze the remaining candidates. Finally, if a reaction matching one of the candidates has been found, then the reaction is removed from the global tuple space. We assume that move commands may not occur in the body of the reaction.

Reductions

$\rightarrow$:

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

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

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

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

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

$\rightsquigarrow$:

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

$$\frac{T \rightsquigarrow_S T'}{T \rightsquigarrow_{v \cup S} T'} \quad \text{(R2)}$$

$$\frac{}{T \rightsquigarrow_{\{\}} T} \quad \text{(R3)}$$

$\Rightarrow$:

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

$$\frac{A\,,\, T\,,\, X \equiv A'\,,\, T\,,\, X' \qquad A'\,,\, T\,,\, X' \;\Rightarrow\; A''\,,\, T'\,,\, X'}{A\,,\, T\,,\, X \;\Rightarrow\; A''\,,\, T'\,,\, X'} \quad \text{(G2)}$$

The rules are subjected to the following side conditions:

(T1) if $v \leq v'$ (T4) $T' = mvt(a, h', T)$

(T2) if $v \leq v'$ (R1) if $T = \langle v'\;\langle a\;h\rangle\;\langle x\;P\rangle\rangle \cup T' \;\wedge v \leq v'$

(T3) $v = mkt(v', a, h, Q)$ (R2) if $\nexists \langle v'\;\langle a\;h\rangle\;P\rangle \in T$ s.t. $v \leq v'$

Table 2. Lime calculus operational semantics

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.

3.2 Restrictions and Extensions.

The semantics presented above can be viewed, in some sense, as an ideal semantics because operations are allowed to operate over the entire federated tuples space and strong atomicity guarantees are enforced throughout. $\text{Lime}_{\mathbf{imp}}$ places three additional restrictions on the calculus. Rather than burdening the semantics with those restrictions we will summarize them here.

R0 In output and input operations, $\mathbf{out}\,\langle v\;a\;s\rangle$ and $\mathbf{in}\,\langle v\;a\;a'\;s\rangle, x$, the tuple space field s can not be the unspecified value. For output operations the destination, a, can not be unspecified.

R1 The body P of a reaction $(\mathbf{react}\,\langle v\;a\;a'\;s\rangle, x.P)$ is not allowed to contain blocking operations such as **in** and **rd**. This restriction prevent reactions from locking up the global tuple space.

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 3. Structure congruence, pattern matching and auxiliary functions

R2 Lime reaction tuples are selected for activation only if the tuple that triggers the reaction is located on the same host as the agent that registered the reaction. In other words, reaction are local to the current host. This restriction avoids having to lock the whole federated tuple space while the reactions are running.

The semantics has omitted some features of Lime. These can be considered as extensions to the basic Lime calculus semantics.

Probes and group operations. Non blocking input operations and operations that return groups of tuples can be defined easily in the formalism. We did not include them not to burden the semantics and syntax of the calculus.

Weak reactions. $\text{Lime}_{\mathbf{imp}}$ includes one kind of distributed reactions, the so-called weak reactions. They are implemented by loosening the atomicity of reaction and introducing an asynchronous step between the identification of the candidate tuples and execution of the reaction. Execution of the reaction remains atomic on the host on which the agent that registered the reaction resides. Weak reactions are implemented by registering strong reaction on every host in the federation. These strong (local) reactions will notify the host that registered a reaction of the insertion of a matching tuple.

Host engagement and disengagement. We chose not to model engagement and disengagement of hosts. Engagement could be modeled by creating a set of new agents running on a fresh host identifier (defined with $(\nu\, h)$). To model disengagement we would have to add a connectivity map that indicate which hosts are connected.

4 Critical Assessment of Lime

During our evaluation we found several inefficiencies in both $\text{Lime}_{\textbf{spec}}$ and $\text{Lime}_{\textbf{imp}}$ which we believe must be addressed if Lime is to gain widespread acceptance. These problems stem from the strong atomicity and consistency imposed by $\text{Lime}_{\textbf{spec}}$. 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 $\text{Lime}_{\textbf{spec}}$ and $\text{Lime}_{\textbf{imp}}$. 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. Empirical experiments were run on a network of PC with a 400 Mhz Dual Pentium II processor using SunOS 5.6 and Sun's VM of JDK 1.2.2. The machines were connected by a 10 Mbits Ethernet network.

4.1 Reaction Livelocks

$\text{Lime}_{\textbf{spec}}$ 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 a 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. In the Lime semantics the following expression $\textbf{react}\, v, x.(!\,\textbf{out}\, v')$ will never terminate as we require that the reaction body reduces to $\mathbf{0}$. In $\text{Lime}_{\textbf{imp}}$ 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, as in the program $\textbf{react}_{\text{once-p-t}}\langle v\ a\ a\ s\rangle, x.\textbf{out}\,\langle v\ a\ s\rangle$. 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 $\text{Lime}_{\textbf{imp}}$. 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 Lime 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 [5]. $\text{Lime}_{\textbf{spec}}$ 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 $\text{Lime}_{\textbf{imp}}$ 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. $\text{Lime}_{\textbf{spec}}$ 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 $\text{Lime}_{\textbf{imp}}$ is to keep a single copy of every tuple on the same host as it's owner agent. Federated input requests 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 operation as the reactions have to be searched for each output.

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.

Experiment 1. We compared the use of remote unicast operations against federated operations by a simple program composed of n agents, each running on a different host and owning one integer, that computed the sum of the values in parallel. The results show that the running time of the version using federated operations is 53% to 88%

higher. Unfortunately, we were not able to scale the experiment to configurations larger than 10 hosts as the current Lime implementation deadlocks after 6 hosts.

Experiment 2. To prove that in Lime agents engaged in a federated tuple space and using only local operations are penalized for remote operations performed by other agents, we wrote a small program in which an agent L_1 consumes tuples produced by another agent L_2 running on the same host. Simultaneously, n remote agents R_i, each running on a different host, consume tuples produced by R_{i+1},except the last one that only produces values. The results show that the speed of local communication between L_1 and L_2 decreases as we increase the number of agents R. With six agents R engaged in the space, L_1 and L_2 start to exchange only 0.18 messages per second, whereas without remote agents, L_1 and L_2 exchange 0.38 messages per second.

4.4 Atomicity of engagements and disengagements

In $\text{Lime}_{\mathbf{imp}}$ hosts joining or leaving a federation must be brought to a consistent state. This boils down to making sure for engagement 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. For the disengage operation, all weak reactions registered by agents currently on that host from all other hosts in the federation have to be de-registered. Since both operations are atomic it means that tuple operations are blocked while hosts are being added or removed from the configuration. 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.

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

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

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

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

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

$mvt(a, h, \{\}) = \{\}$
$mvt(a, h, \langle v\;a\;a'\;s\rangle \cup T) = \langle v\;a'\;a'\;s\rangle \cup mvt(a, h, T)$, if $loc(a') = h$
$mvt(a, h, \langle v\;a'\;a\;s\rangle \cup T) = \langle v\;a\;a\;s\rangle \cup mvt(a, h, T)$, 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 4. Semantics of CoreLime.

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 4.

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, where 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. The auxiliary function *react* will create a single new agent on the current host with as body the parallel

composition of matching reactions. 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 r to another host h' to perform a remote **in** . 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.

$$
\begin{aligned}
&a_h[\,\mathbf{rin}\ h', \langle v', a', a'', s\rangle, x.P \mid P'\,] \triangleq \\
&\quad (\nu\, y)\,(\nu\, r)\ \ a_h[\,\mathbf{in}\,\langle\langle y, ?\rangle, a, a, s\rangle, x.P \mid P'\,] \mid \\
&\quad r_h[\,\mathbf{move}\ h'.\mathbf{in}\,\langle v', a', a'', s\rangle, x.\mathbf{move}\ h.\mathbf{out}\,\langle\langle y, x\rangle, a, s\rangle\,]
\end{aligned}
$$

5.2 Security extensions for CoreLime

The security extensions described here are being incorporated in our implementation of CoreLime. Unless otherwise specified we assume that the $\text{Lime}_{\mathbf{imp}}$ interfaces have been retained. We propose two new mechanisms for adding support for security into Lime. The first is a capability-based access control model for controlling access to tuple spaces. The second is an extension of the concept of reactions called filters.

Fine-grained access control Fine-grained access control mechanisms provide applications the means to control access to shared tuple spaces. For instance, it may be desirable to restrict some applications to have write-only access, i.e. they can add tuples to a space but not remove them, or conversely, they may be allowed to input tuples but not to add new ones. Reactions may be even more sensitive than other operations as a process than can register reactions may inspect all tuples put in the space. The central idea is one inspired from Cardelli's Ambients in which ambient names behave as capabilities and can be given out in restricted form.

In CoreLime, each tuple space has a name, an instance of the abstract interface `CoreLimeName`, see Figure 1. Names can always be compared for equality, with the same purpose as in Lime, that is, spaces with the same name can be shared. But in addition names are used by CoreLime to implement access control. Each name acts as a capability and is checked to validate every operation. CoreLime provides two implementations for `CoreLimeName`, one is the class `LimeName` which is provided for backwards compatibility with Lime programs. Instances of `LimeName` contain a string and do not implement any access control.

The class `CoreLimeCapability` implements the full functionality of the interface `CoreLimeName`. Each capability contains a globally unique identifier generated when the object is created and hidden from user code. The `isSame` method compare GUIDs for equality. Since user code can not see the name of the tuple space, in order to create shared spaces there must be some name exchange protocol in place. The agent that creates a name will hand out copies of that name

```
interface CoreLimeName {
    boolean isSame(CoreLimeName); // Compare two names.
    CoreLimeName clone();       // Return a name with same rights.
    void forbidRead();  // Remove the right to remove tuples from space.
    void forbidWrite(); // Remove the right to insert tuples in space.
    void forbidRegister(); // Remove the right to register reactions.
    void forbidClone();         // Remove the right to clone this name.
}
final class LimeName implements CoreLimeName {
    LimeName(String name);
}
final class CoreLimeCapability implements CoreLimeName {
    CoreLimeCapability();
}
```

Fig. 1. CoreLime Capability interface.

to other agents after having set the access rights associated to the name to the proper level. For instance, the following code fragment creates a new name and outputs a copy of that name which can not be further distributed and which forbids writes to the tuple space.

```
CoreLimeName privateKey = new CoreLimeCapability(); // unrestricted name
CoreLimeName publicKey = privateKey.clone();
publicKey.forbidClone(); // publicKey can not be copied
publicKey.forbidWrite(); // publicKey can not be used to out
templ = new Tuple();
templ.addActual(publicKey);
ts.out(templ);
```

The implementation of CoreLime ensure that non-cloneable keys can not be copied and that only one instance can be serialized. Whenever names are serialized the CoreLime implementation will digitally sign the name to prevent tampering during transit. The interface to names only allows capabilities to be removed, thus if we write $publicKey.forbidRegister()$ it means that this particular capability will never again allow a reaction to be registered for the particular tuple space.

Security filters Security filters are a special kind of reaction that can be installed on a tuple space to perform some actions at each Lime operation. The goal of filters is to allow even finer-grained security policies to be coded such as policies that look at the values of tuples being inserted or extracted from the space. Filters get as input the tuple given as argument to the Lime operation and may return a modified tuple or `null` if the operation should fail. Multiple filters can be defined for the same space and operations. They will be chained in an implementation specific order.

Consider for example an output filter that checks that first field of every output tuple bears an agent name.

```
class AgNmFilter extends CoreLimeFilter {
    ITuple filterOutput(ITuple val) {
            if (val.get(0) instanceof AgentLocation)
                return val;
            else return null };
}
```

Further checking can be enforced by adding another filter that appends the name of the agent that produced the tuple to each value being output.

```
class AccFilter extends CoreLimeFilter {
    ITuple filterOutput(ITuple val) {
        val.addFormal(CoreLime.getCurrentAgentName());
        return val; }
    }
```

The above filters can be chained in the following manner:

```
ts.registerFilter(new AccFilter());
ts.registerFilter(new AgNmFilter());
```

Filter expression are owned by agents, just as reactions. They move with them and are merged when a space is transiently shared.

One important difference between reactions and filters is that filters can be applied to input expressions as well as to the registering of reactions. For example, it is possible to enforce that an agent will only be able to *remove* the tuples it has inserted with the following input filter:

```
class InputAccFilter extends CoreLimeFilter {
    ITuple filterIn(ITuple val) {
        val.addFormal(CoreLime.getCurrentAgentName());
        return val };
}
```

This filter is symmetric to the `AccFilter` in that it appends the name of the agent to all input requests currently processed by the CoreLime system.

6 Conclusions

This paper revisited Lime, a middleware communication infrastructure for mobile computation that addresses both physical mobility of devices as well as logical mobility. We have argued that the original Lime specification is costly and complex to implement. We have proposed a smaller and lighter variant of Lime, which we call CoreLime, that has none of the global synchronization and atomicity requirements of Lime. Furthermore we have presented the access control mechanisms built into our extension the Java implementation of Lime. We are currently working towards a production quality implementation of CoreLime. In future work, we plan to translate the distributed features of Lime into our simpler model.

Acknowledgments This research was supported by a grant from Motorola, CERIAS and CAPES. We wish to thank the authors 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 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.
4. N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, Feb. 1998.
5. 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.
6. 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.
7. 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.
8. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.
9. 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.
10. D. B. Lange and M. Oshima. Mobile agents with Java: The Aglet API. *World Wide Web Journal*, 1998.
11. J.-H. Morin. HyperNews: a Hyper–Media Electronic Newspaper based on Agents. In *Proceedings of HICSS-31, Hawai International Conference on System Sciences*, pages 58–67, Kona, Hawaii, January 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. 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.
15. J. Vitek. *The Seal model of Mobile Computations.* PhD thesis, University of Geneva, 1999.