Math. Struct. in Comp. Science (2004), *vol.* 14, *pp.* 397–419.
DOI: 10.1017/S0960129504004177 Printed in the United Kingdom

Coordination and mobility in CoreLime

BOGDAN CARBUNAR[†], MARCO TULIO VALENTE[‡]
and JAN VITEK[†§]

[†]*Dept. of Computer Sciences, Purdue University, USA*
Email: carbunar@cs.purdue.edu

[‡]*Universidade Federal de Minas Gerais, Brazil*

Received 9 March 2002; revised 20 April 2003

The choice of suitable high-level communication primitives for wide area network programming languages remains an open problem. This paper is driven by the practical consideration of providing an efficient and secure communication infrastructure for mobile agent systems. This has led us to formalise the Lime coordination middleware and propose a simplified model, which we call CoreLime, that addresses some of the main shortcomings of Lime while retaining its distinguishing feature, namely *transient sharing of tuple spaces*. We further discuss a prototype implementation along with security extensions. Our contribution is thus an exploration of the language design space rather than a theoretical investigation of properties of these models.

1. Introduction

Traditional computational models are based on the assumption that software and the devices it runs on are deployed before being used, and that, once deployed, software configurations remain static. Wireless and *ad hoc* networks challenge this assumption by suggesting computational models that exploit the dynamic nature of the physical and logical infrastructure. This has motivated a line of research on languages in which mobility is the central linguistic abstraction (Cardelli and Gordon 1998; Vitek 1999; Fournet and Gonthier 1996; Wojciechowski and Sewell 2000). While these languages provide the means by which applications can have some degree of control over locality, they typically fail to provide expressive communication primitives.

In previous work, we developed a medium-sized application on top of a mobile agent system (Morin 1998; Bryce and Vitek 1999). While the application software was faced with many of the traditional challenges of distributed computing, surprisingly, the majority of the code was devoted to the implementation of inter-agent communication protocols. These had to be built upon the channel abstraction provided by the agent infrastructure (Vitek 1999). In retrospect, the difficulties were predictable. The services offered by the agent infrastructure were limited to local communication and naming. The agent system had no provision for distributed communication and provided only the simplest of communication mechanisms, namely synchronous named channels, as in the π-calculus

[§] This work was supported by NSF under grant CCR-0093282.

(Milner *et al.* 1992). Thus *ad hoc* schemes had to be developed in our application to support resource discovery, distributed naming, agent authentication, distributed messaging. Even then, agent communication tended to be cumbersome. Of course, there are good reasons for choosing low-level primitives. The whole point of mobile languages is to emphasise locality, and distributed channels, as in the Join calculus (Fournet and Gonthier 1996), would defeat the purpose. The other justification is that from a theoretical point of view, high-level communication mechanisms are not needed since they can be encoded (Wojciechowski and Sewell 2000).

In practice, mobile systems do require high-level communication mechanisms. In our experience, interaction in mobile system has the following distinctive characteristics:

— **Transient and Opportunistic:** Communication patterns are shaped by the nature of an environment in which hosts are intermittently connected and agents can migrate at any time. Communication is opportunistic as applications take advantage of resources that happen to be available at a given time without relying on their continued availability. Communication protocols must accommodate long latencies and timeouts caused by the sudden departure of an interlocutor or disconnection of the agent itself.

— **Anonymous and Untrusted:** Interactions are based on services offered rather than on the identity of the entity providing those services. Agents do not need to know each others names or locations, they simply require particular services. Anonymity's corollary is that agents are not necessarily trusted and require support for the implementation of secure communication protocols.

The spatial and temporal uncoupling provided by tuple space-based languages (Gelernter 1985) is well suited to the communication patterns of mobile systems. In Bryce *et al.* (1999), we investigated coordination languages as a replacement for the channels of our agent infrastructure (Bryce and Vitek 1999). In particular, we tried to add security to a Linda-like model. However, some of the solutions presented in that work were unsatisfactory as they did not account for agent mobility. This meant that unconsumed messages, traditionally called tuples, would never be reclaimed. This issue was left as future work as no semantically palatable solution could be found.

The *Lime* middleware infrastructure of Murphy *et al.* (Picco *et al.* 1999) provides a partial answer. Lime is an elegant combination of Linda with reactive programming designed to provide a simple communication model for mobile environments. Lime introduces the notion of *transiently shared tuple spaces*: each mobile entity is equipped with its own individual tuple space, which moves whenever that entity moves. These individual tuple spaces are silently merged 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. 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.

Our original goal was simply to extend Lime with the access control mechanisms needed to implement secure interaction between untrusted parties (Bryce *et al.* 1999) and use that model in the implementation of a new mobile agent system for limited capacity connected devices. Along the way we realised 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. Section 2 gives a brief informal overview of Lime. In Section 3 we provide a formalisation of the core concepts of Lime as a process calculus. The semantics gives a well understood starting point for reasoning about Lime programs and, more importantly for our needs, was meant to serve as a specification for our implementation. This process revealed some rather serious shortcomings of the model, which are outlined in Section 4. Faced with these issues, we decided to simplify the model at the cost of some expressiveness, and defined CoreLime, an even simpler calculus, which does not have the inherent inefficiencies of Lime. The semantics of CoreLime is given in Section 5. Finally, we describe security extensions that we are adding to our implementation of CoreLime.

2. Middleware for mobile environments

The Lime middleware is a communication infrastructure for mobile environments written in the Java programming language and introduced in Picco *et al.* (1999). Large parts of the model were formalised in Mobile Unity notation in Murphy (2000). In this section we present an overview of the main concepts of Lime. When necessary, we will differentiate between the implementation (denoted Lime_{imp}) and its specification (denoted $\text{Lime}_{\text{spec}}$).

Basics Lime programs 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. Agents can deposit a datum 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`. The last two operations do not modify the tuple space. Figure 1 demonstrates the Lime implementation of a producer/consumer protocol with two agents exchanging data over a common space. The producer thread repeatedly creates a tuple containing two actuals, the first of which is a string and the other an integer value, using the Lime `addActual` operation (lines 2 and 5). The producer proceeds to output the tuple to the tuple space `ts` (line 6). The consumer extracts the tuples from the same space using the `in` primitive (line 13). The input operation takes a query tuple, which, in this case, consists of two values, an actual that matches the string used by the producer and a formal that restricts the query to tuples with an integer as second value (lines 9–11). The tuples are retrieved non-deterministically depending on the interleaving of the threads.

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

producer

```
01. tuple = new Tuple();
02. tuple.addActual("key");

03. for (int i = 0; i < 10 ; i++) {
04.     val = tuple.copy();
05.     val.addActual(new Integer(i));
06.     ts.out(val);
07. }
```

consumer

```
08. tuple = new Tuple();
09. tuple.addActual("key");
10. templ = tuple.copy();
11. templ.addFormal(Integer.class);

12. for (int i = 0; i < 10 ; i++)
13.     tuples[i] = ts.in(templ);
```

Fig. 1. Producer/Consumer in Lime.

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. So, an `out` operation can be thought of as taking two steps. Assuming that an agent *bob* outputs a tuple destined for another agent *alice*, the first step will be to emit the tuple with a destination of *alice* and owner of *bob*. Then, in a second step, if *alice* is reachable, ownership of the tuple can be transferred to *alice*:

$$\textbf{bob}\colon \texttt{ts.out(alice, tuple)} \rightarrow \langle \textbf{bob}, \textbf{alice}, \texttt{tuple}\rangle \rightarrow \langle \textbf{alice}, \textbf{alice}, \texttt{tuple}\rangle$$

A tuple for which `current` $\neq$ `final` is in transit (also called *misplaced* in Lime). Lime implementations need not maintain these fields explicitly, but they are useful for the exposition. Input operations have two location parameters corresponding to the above mentioned fields. Thus, an input operation with the following query template `(current, final, template)` means that the Lime implementation will search for tuples matching `template` in the space of agent `current` having agent `final` as destination. Lime allows either parameter to be unspecified in order to broaden the scope of the query. Finally, Lime_{imp} allows us to specify host identifiers as the current location.

Reactive programming In addition to the standard Linda primitives, Lime supports *reactions*. A reaction can be viewed as a triple `(t, s, p)` consisting of a tuple space reference `t`, a template `s` and a code fragment `p`. The semantics of a reaction is that

```
01. templ = new Tuple();
02. templ.addActual("example");
03. templ.addFormal(Integer.class);

04. listener = new ReactionListener {
05.     int count;
06.     public void  reactsTo(ReactionEvent e) {
07.         System.out.println("reaction " +
08.             ++count + " fired");
09.     }};

10. reactions[0] = new LocalisedReaction(
11.                                 currentHost,
12.                                 thisAgent,
13.                                 templ,
14.                                 listener,
15.                                 Reaction.ONCEPERTUPLE);
16. ts.addStrongReaction(reactions);
```

Fig. 2. Reactions example following Figure 1.

whenever a tuple matching `s` is deposited in `t`, the code fragment `p` should be run. The main difference between blocking `rd` and reactions is that *all* 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* tuple 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 producer/consumer example of Figure 1, another thread may register a reaction that prints a message on the console each time a tuple bearing string `example` is inserted in the space (see Figure 2). In this example, we create a query template `templ` that will match the appropriate tuples (lines 1–3). The code of the reaction is encapsulated in an anonymous class (lines 4–9). The variable `count` is used to keep track of the triggered reactions. The reaction is then created with a once per tuple modality that ensures that it will fire for each tuple deposited in the tuple space `ts` by the producer thread (lines 10–15).

Transiently Shared Spaces By default, the tuple spaces of different agents are disjoint. 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 other agents, that have 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. The assumption is that

Agent1

```
01. tuple = new Tuple();
02. tuple.addActual(new Integer(1));
03. ts.out(Agent2, tuple);
// deposit ⟨Agent1, Agent2, tuple⟩ in ts
```

Agent2

```
04. templ = new Tuple();
05. templ.addActual(new Integer(1));
06. tuple = ts.in(Agent2, Agent2, templ);
    // block (line 06)
    // ... move to Agent1.host
    // deliver ⟨Agent2, Agent2, tuple⟩ and unblock (line 06)
```

Fig. 3. Transiently shared spaces.

agents can leave a host at any time. When this occurs, Lime will break up the tuple space and extract all tuples that have the departing agent in their `current` field. Consider the example of Figure 3 in which two agents own a tuple space `ts`. This tuple space becomes transiently shared if one of the agents migrates to the same host as the other. When this occurs, the tuple output by the first agent (line 03) can be read by the second (line 06). Transient sharing simplifies the coding of application communication protocols as explicit location parameters can often be omitted. Thus in the above example, the second agent could have emitted a simple `tuple = ts.in(templ)` to retrieve the same tuple. Of course, omitting location parameters implies tuples emitted by other agents become eligible for input operations.

Federated Spaces 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 is 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, Lime_{imp} introduces weak reactions and limits (strong) reactions to a single host. A weak reaction may be scoped over multiple hosts, but it adds an asynchronous step between the 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 Lime_{imp}, a weak reaction is implemented by registering one strong reaction on every node of the federation.

Summary This concludes our overview of Lime. Interested readers should refer to Murphy *et al.* (2001) for a detailed presentation. We now turn to a formal presentation of the key features of Lime.

Table 1. *Lime calculus syntax.*

$$
\begin{array}{rcll}
Prog & ::= & A\ \mathcal{N} & \\
A & ::= & \varepsilon \mid a_h[P] \mid A & \\
P & ::= & \mathbf{0} \mid P \mid Q \mid\ !P \mid (\nu\, x)\, P \mid & \\
 & & \mathbf{out}\, v \mid \mathbf{in}\, v, x.P \mid \mathbf{rd}\, v, x.P \mid \mathbf{react}\ v, x.P \mid \mathbf{move}\ h.P & \\
 & & & \\
v & ::= & N \mid & \textit{names} \\
 & & P \mid & \textit{processes} \\
 & & \langle v_1 \ldots v_n \rangle & \textit{tuples}
\end{array}
$$

3. The Lime Calculus

Syntactically, Lime could be construed as an extension of Linda with **move** and **react** operations. The meaning of Lime's primitives is, however, quite different. To understand these differences we turn to a formal description. As Murphy's formalisation of Lime based on Chandy and Misra's UNITY necessitates numerous extraneous definitions and is rather complex, we decided to present the operational semantics of Lime as a process calculus modelled after the asynchronous π-calculus. The main difference between our formalism and 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 (Busi *et al.* 1998; DeNicola and Pugliese 1996). The advantages of this approach is that the semantics is self-contained, simple and enjoys broad acceptance.

Table 1 presents the syntax of the Lime calculus. We assume a set of *names* N ranged over by meta-variables, a, s, h, x, y. *Values*, ranged over by v, consist of names, processes, and tuples. *Tuples* are ordered sequences of values $\langle v_1 \ldots v_n \rangle$. We use the symbol $\star$ to denote the distinguished unspecified value. As usual, this value is used to broaden the scope of matching operations. A *configuration* $A\ \mathcal{N}$ is defined as the parallel composition of a set of agents and an environment $\mathcal{N}$ consisting of a multiset of tuples $\mathcal{N}_T$, and a set of names $\mathcal{N}_X$, and it is also written $\mathcal{N}_T \mathcal{N}_X$. An agent $a_h[P]$ has a unique name a, runs on a host h and its behaviour is described by the process P. Processes are the engines that drive agents. They are ranged over by metavariables P and Q. The first four process kinds are borrowed from the asynchronous π-calculus. The inert process $\mathbf{0}$ exhibits no behaviour. 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. Restriction operators are applicable within an agent only. To communicate names between agents, these names must be added to the global environment using the following equivalence on configurations:

$$a_h[(\nu\, x)\, P] \mid A\ \ \mathcal{N}_T \mathcal{N}_X \ \equiv\ a_h[P] \mid A\ \ \mathcal{N}_T \mathcal{N}_X \oplus x$$

under the assumption that $x \notin \mathcal{N}_X \cup fn(A)$. The remaining process kinds are primitive operations. The $\mathbf{out}\, v$ operation takes a tuple $\langle v\ a\ s \rangle$, v is the value to be inserted in tuple

space s of the destination agent a. In the sample configuration

$$a_h[\ \mathbf{out}\ \langle x\ a'\ s\rangle\ ]\ \mid\ a'_h[\ P\ ]\quad \mathcal{N}$$

the output tuple will be sent to tuple space s of agent a' with x as payload. We impose a well-formedness constraint on output expressions: they are not allowed to contain occurrences of the unspecified value $\star$. The operation has no continuation as output is asynchronous in Lime. Operations $\mathbf{in}\, v, x.P$ and $\mathbf{rd}\, v, x.P$ take three arguments a tuple v, a name x to bind the result of the operation as arguments, and a continuation process P. The tuple v has the shape $\langle v'\ a\ a'\ s\rangle$ where v' is the query template describing the shape of the tuple to match, a and a' are the current owner and final destination agents of the desired tuple, and s is the name of the target tuple space. Unspecified values may be used to broaden the scope of inputs, for example, if both current and destination fields are left unspecified, the entire space will be searched as shown in the configuration

$$a_h[\ \mathbf{out}\ \langle x\ a'\ s\rangle\ ]\ \mid\ a'_h[\ \mathbf{in}\ \langle x\ \star\ \star\ s\rangle, y\ ]\quad \mathcal{N}$$

where the entire tuple space s is searched for tuples carrying x. The **move** primitive allows agents to change the host on which they are executing. When Lime processes a move request, tuples belonging to the migrating agent are atomically removed from the source host and inserted into the destination's tuple spaces. In the configuration

$$a_h[\ \mathbf{move}\ h'\ ]\quad \{\langle v\ a\ a\ s\rangle\}\mathcal{N}_X,$$

moving agent a will cause its location to be changed to h' and its single tuple to be moved to h'. As tuple locations are implicit in the semantics, the environment does not need to be modified during the move. The **react** operation allows an agent to register its interest in events. Three arguments are expected: the template to match the event, similar to **in**'s first argument, x, which will be bound to the actual event, and P, which is the event handler. The reaction operation in the following configuration will be triggered whenever a one-element tuple is inserted in space s

$$a_h[\ \mathbf{react}\ \langle\langle\star\rangle\ \star\ \star\ s\rangle, x\ .\ \mathbf{out}\ \langle x, a', s'\rangle\ ]\quad \mathcal{N}.$$

The behaviour of the reaction is to output a copy of the tuple in the space s' of agent a'. Agents may access tuple spaces by name, thus names act as capabilities. In our formalisation tuple spaces are represented by disjoint subsets of the global tuple multiset. Every tuple output by a process is stored in $\mathcal{N}_T$ as

$$\langle v\ a\ a'\ s\rangle$$

where v is a tuple containing the payload of the communication, and a, a' and s encode routing and location information: s is the tuple space name, a is the name of the agent currently hosting the tuple and a' is the name of its intended destination agent. Consider the configuration

$$a_h[\ \mathbf{out}\ \langle x\, a\, s\rangle\ \mid\ \mathbf{out}\ \langle x'\, a'\, s\rangle\ ]\quad \mid\quad a'_{h'}[\ \mathbf{out}\ \langle y\, a\, s\rangle\ \mid\ \mathbf{move}\, h\ ]\quad \{\}\mathcal{N}_X.$$

Assuming that the output operations are scheduled before the move, the environment would be extended to $\{\langle x\, a\, a\, s\rangle, \langle x'\, a\, a'\, s\rangle, \langle y\, a'\, a\, s\rangle\}\mathcal{N}_X$. After the move, all tuples will be delivered and the environment will be $\{\langle x\, a\, a\, s\rangle,\ \langle x'\, a'\, a'\, s\rangle, \langle y\, a\, a\, s\rangle\}\mathcal{N}_X$.

For simplicity we have omitted operations for extracting values from tuples, these can be defined in the obvious way (Bryce *et al.* 1999).

3.1. *Semantics of Lime*

Table 2 presents the operational semantics of Lime as a reduction semantics. Our semantics can be viewed, in some sense, as an ideal semantics because operations are allowed to affect the entire federated tuple space and strong atomicity guarantees are enforced throughout.

For readability we use the notation $\mathscr{E}[\![P]\!]$ as a shorthand for $a_h[\,P\,] \mid A$. Furthermore, $S \oplus x$ is a shorthand for $S \cup \{x\}$, and P is a shorthand for $P.\mathbf{0}$. We work up to alpha conversion of bound names throughout, writing the free name function, defined in Table 3, as $fn\ \_$. Finally, we define $(fold\ f\ op\ z)\{a_0, \ldots, a_n\} = (f\ a_0)\ op\ \ldots\ op\ (f\ a_n)\ op\ z$. To simplify the presentation, the operational semantics is split into three sets of reduction rule, which we present next.

Primitive operations The rewrite rules for the basic operations have the form $A\ \mathscr{N} \rightarrow A'\ \mathscr{N}'$. Each step of reduction represents the effect on the program and tuple space of executing one Lime primitive operation. The first two rules (T1–T2) model the behaviour of **in** and **rd** operations. They succeed if there is a tuple matching the argument template, in which case every free occurrence of x in the continuation is replaced by the matched tuple. The **in** rule is

$$\mathscr{E}[\![\mathbf{in}\, v, x.P \mid P']\!]\ \ \mathscr{N}_T \mathscr{N}_X\ \ \rightarrow\ \ \mathscr{E}[\![P\{v'/x\} \mid P']\!]\ \ \mathscr{N}'_T \mathscr{N}_X$$

where $\mathscr{N}'_T$ is $\mathscr{N}_T$ without the matched tuple. If multiple tuples match the template, one is chosen randomly. The definition of pattern matching, written $v \leqslant v'$, allows for recursive tuple matching as in $\langle\langle\star\rangle\ \star\rangle \leqslant \langle\langle x\rangle\ \langle x\rangle\rangle$.

The **out** rule (T3) inserts a tuple in an agent's tuple space.

$$\mathscr{E}[\![\mathbf{out}\, v' \mid P]\!]\ \ \mathscr{N}_T \mathscr{N}_X\ \ \rightarrow\ \ \mathscr{E}[\![P]\!]\ \ \mathscr{N}'_T \mathscr{N}_X$$

The tuple space $\mathscr{N}'_T$ extends $\mathscr{N}_T$ with a tuple generated by the function *mkt*, defined in Table 3, such that the output value $\langle v\ a'\ s\rangle$ is transformed into a tuple $\langle v\ x\ a'\ s\rangle$, where x is either a, the current agent known as the owner, or a', the destination agent. If the destination is in the configuration, the tuple is delivered and its owner is set to a'. If the destination agent is not present, the tuple is *misplaced* and a retains ownership.

The **react** reduction modifies the global tuple space by adding a reaction tuple $\langle v\ \langle a\ h\rangle\ \mathbf{in}\, v, x.P\rangle$. The fields of the tuple are the template v that triggers the reaction, the name a of the agent that registered it and its current location h, and the reaction handler $\mathbf{in}\, v, x.P$. As an example, the configuration

$$a_h[\,\mathbf{react}\,\langle\star\ a'\ a''\ s\rangle, x.P_1 \mid P_2\,]\ \mid a'_{h'}[\,\mathbf{out}\,\langle\langle x\rangle\ a''\ s\rangle \mid Q\,]\ \ \mathscr{N} \qquad \text{(CF1)}$$

can take a step and evolve into

$$a_h[\,P_2\,]\ \mid a'_{h'}[\,\mathbf{out}\,\langle\langle x\rangle\ a''\ s\rangle \mid Q\,]\ \ \mathscr{N}'_T \mathscr{N}_X \qquad \text{(CF2)}$$

where $\mathscr{N}'_T = \mathscr{N}_T \oplus \langle\langle\star\ a'\ a''\ s\rangle\ \langle a\ h\rangle\ \mathbf{in}\, v, x.P_1\rangle$.

Finally, the **move** rule changes the location of the agent and moves all its tuples, including reactions, to the destination host. The tuple relocation is performed by the *mvt* function.

Table 2. *Lime calculus operational semantics.*

$$\mathscr{E}[\![\mathbf{in}\, v, x.P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P\{v'/x\} \mid P']\!]\ \mathscr{N}' \quad \text{(T1)}$$

$$\mathscr{E}[\![\mathbf{rd}\, v, x.P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P\{v'/x\} \mid P']\!]\ \mathscr{N} \quad \text{(T2)}$$

$$\mathscr{E}[\![\mathbf{out}\ v' \mid P]\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P]\!]\ \mathscr{N}' \quad \text{(T3)}$$

$$\mathscr{E}[\![\mathbf{react}\ v, x\,.\,P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P']\!]\ \mathscr{N}' \quad \text{(T4)}$$

$$a_h[\mathbf{move}\ h'\,.\,P \mid P'] \mid A\ \mathscr{N} \rightarrow a_{h'}[P \mid P'] \mid A\ \mathscr{N}' \quad \text{(T5)}$$

$$\frac{a_h[P\{v'/x\}]\ \mathscr{N}' \Rightarrow^* a_h[\mathbf{0}]\ \mathscr{N}'' \qquad \mathscr{N}'' \rightsquigarrow_{v \cup S} \mathscr{N}'''}{\mathscr{N} \rightsquigarrow_{v \cup S} \mathscr{N}'''} \quad \text{(R1)}$$

$$\frac{\mathscr{N} \rightsquigarrow_S \mathscr{N}'}{\mathscr{N} \rightsquigarrow_{v \cup S} \mathscr{N}'} \quad \text{(R2)} \qquad \frac{}{\mathscr{N} \rightsquigarrow_{\{\}} \mathscr{N}} \quad \text{(R3)}$$

$$\frac{\mathscr{N}' \rightsquigarrow_S \mathscr{N}'' \qquad A\ \mathscr{N} \rightarrow A'\ \mathscr{N}'}{A\ \mathscr{N} \Rightarrow A'\ \mathscr{N}''} \quad \text{(G1)} \qquad \frac{A\ \mathscr{N} \equiv A'\ \mathscr{N}' \qquad A'\ \mathscr{N}' \Rightarrow A''\ \mathscr{N}''}{A\ \mathscr{N} \Rightarrow A''\ \mathscr{N}''} \quad \text{(G2)}$$

The rules are subject to the following side conditions:

(T1-2) $\mathscr{N} = \mathscr{N}'_T \oplus v'\mathscr{N}'_X \ \wedge\ v \leqslant v'$
(T3) $\mathscr{N}' = \mathscr{N}_T \oplus v\mathscr{N}_X \ \wedge\ v = mkt\ v'\, a\, h\, A$
(T4) $\mathscr{N}' = \mathscr{N}_T \oplus r\mathscr{N}_X \ \wedge\ r = \langle v\ \langle a\ h\rangle\ \langle x\ P\rangle\rangle$
(T5) $\mathscr{N}' = (mvt\ a\, h'\, \mathscr{N}_T)\mathscr{N}_X$
(R1) $\mathscr{N}'_X = \mathscr{N}_X \oplus a\ \wedge\ a \notin \mathscr{N}_X\ \wedge\ \mathscr{N}_T = \mathscr{N}'_T \oplus \langle v'\ \langle a'\ h\rangle\ \mathbf{in}\, v', x.P\rangle\ \wedge\ v' \leqslant v$
(R2) $\nexists v', \langle v'\ \langle a\ h\rangle\ P\rangle \in \mathscr{N}_T \wedge v' \leqslant v$
(G1) $S = \mathscr{N}'_T - \mathscr{N}_T$

Structural Congruence Rules

$$P \mid Q \equiv Q \mid P \quad \text{(SC1)}$$
$$!P \equiv P \mid !P \quad \text{(SC2)}$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad \text{(SC3)}$$
$$P \mid \mathbf{0} \equiv P \quad \text{(SC4)}$$
$$(v\, x)(v\, y)\, P \equiv (v\, y)(v\, x)\, P \quad \text{(SC5)}$$
$$P \equiv Q \Rightarrow (v\, x)\, P \equiv (v\, x)\, Q \quad \text{(SC6)}$$
$$(v\, x)(P \mid Q) \equiv P \mid (v\, x)\, Q \quad \text{(SC7)}$$
$$a_h[P] \mid A\ \mathscr{N} \equiv A \mid a_h[P]\ \mathscr{N} \quad \text{(SC8)}$$
$$a_h[(v\, x)\, P] \mid A\ \mathscr{N} \equiv a_h[P] \mid A\ \mathscr{N}' \quad \text{(SC9)}$$
$$P \equiv Q \Rightarrow \mathscr{E}[\![P]\!]\ \mathscr{N} \equiv \mathscr{E}[\![Q]\!]\ \mathscr{N} \quad \text{(SC10)}$$

The congruence rules are subjected to the following side conditions:
(SC7) $x \notin fn\, Q$
(SC9) $x \notin fn\, A \ \wedge\ x \notin \mathscr{N}_X \ \wedge\ \mathscr{N}' = \mathscr{N}_X \oplus x\mathscr{N}_T$

Pattern Matching Rules

$$x \leqslant x \qquad \star \leqslant x \qquad \frac{v_1 \leqslant v'_1\ \ldots\ v_n \leqslant v'_n}{\langle v_1\ \ldots\ v_n\rangle \leqslant \langle v'_1\ \ldots\ v'_n\rangle}$$

Table 3. *Auxiliary functions.*

Functions

$$
\begin{array}{lll}
mkt\ \ \langle v\ a'\ s\rangle\, a\, h\, A & = \langle v\ a'\ a'\ s\rangle, & \text{if } A \equiv a'_{h'}[P] \mid A' \\
mkt\ \ \langle v\ a'\ s\rangle\, a\, h\, A & = \langle v\ a\ a'\ s\rangle, & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
mvt\ \ a\, h & = (fold\ (mvt'\, a\, h) \cup \{\}) & \\
mvt'\ \ a\, h\, \langle v\ \langle a\ h'\rangle\ k\rangle & = \langle v\ \langle a\ h\rangle\ k\rangle & \\
mvt'\ \ a\, h\, v & = v, & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rlrl}
fn\ x & = \{x\} & fn\ \langle\rangle & = \{\} \\
fn\ \langle v_0 \ldots v_n\rangle & = fn\ v_0 \cup \ldots \cup fn\ v_n & fn\ P \mid Q & = fn\ P \cup fn\ Q \\
fn\ !P & = fn\ P & fn\ 0 & = \{\} \\
fn\ (v\ x)\, P & = fn\ P - x & fn\ \mathbf{out}\, v & = fn\ v \\
fn\ \mathbf{in}\, v, x.P & = fn\ P - x \cup fn\ v & fn\ \mathbf{rd}\, v, x.P & = fn\ P - x \cup fn\ v \\
fn\ \mathbf{react}.P & = fn\ P - x \cup fn\ v & fn\ \mathbf{move}\ h.P & = fn\ P \oplus x
\end{array}
$$

Reaction rules The second set of reduction rules models the execution of reactions. Since reactions are kept as special types of tuples in the environment, the rules have the format $\mathcal{N} \rightsquigarrow_S \mathcal{N}'$, where S is the set of tuples candidate for triggering reactions. There are three rules for evaluating reactions. Rule (R3) simply states that when the set S is empty, there are no reactions to execute. When the set is not empty but there is a tuple that does not match any reaction, the tuple is discarded using (R2):

$$\frac{\mathcal{N} \rightsquigarrow_S \mathcal{N}'}{\mathcal{N} \rightsquigarrow_{v \cup S} \mathcal{N}'}.$$

Rule (R1) deals with the more interesting case when a tuple triggering a reaction is found:

$$\frac{a_h[P\{v'/x\}]\ \mathcal{N}' \ \Rightarrow^*\ a_h[\mathbf{0}]\mathcal{N}'' \qquad \mathcal{N}'' \rightsquigarrow_{v \cup S} \mathcal{N}'''}{\mathcal{N} \rightsquigarrow_{v \cup S} \mathcal{N}'''}.$$

A new agent with a fresh name is then created to run on the host where the owner of the reaction currently resides, and that evaluates the reaction handler with every free occurrence of x replaced by the matching tuple. The new agent executes in an environment that no longer contains the current reaction but is extended with the name of the agent. The evaluation of the agent is done using the rule $\Rightarrow$, which is described in the next paragraph. The resulting $\mathcal{N}''$ is used to evaluate the remaining reactions recursively, under the same set of candidate tuples, since the current matching tuple might trigger other reactions.

Global computations The final set of rules combines the previous two. The rule format is $A\ \mathcal{N} \Rightarrow A'\ \mathcal{N}'$. (G1) shows that after every primitive step all the triggered reactions have to be evaluated

$$\frac{\mathcal{N}' \rightsquigarrow_S \mathcal{N}'' \qquad A\ \mathcal{N}\ \rightarrow\ A'\ \mathcal{N}'}{A\ \mathcal{N}\ \Rightarrow\ A'\ \mathcal{N}''}$$

where $S = \mathcal{N}'_T - \mathcal{N}_T$. Lime allows reactions only for the insertion of tuples, so only **out** and **move** are of real interest for the execution of reactions. This is illustrated in the content of S, which is the set of tuples added or modified by the primitive step. In rule

(G2) we make use of structural congruence rules to take a global step:

$$\frac{A\ \mathcal{N} \equiv A'\ \mathcal{N}' \qquad A'\ \mathcal{N}'\ \Rightarrow\ A''\ \mathcal{N}''}{A\ \mathcal{N}\ \Rightarrow\ A''\ \mathcal{N}''}.$$

The rule states that if a configuration is structurally congruent with a configuration that takes a step, then the initial configuration takes the same step. We impose an additional well-formedness constraint on configurations: the names of all agents and hosts must be included in the environment's set of names $\mathcal{N}_X$.

Structural congruence Table 2 presents the congruence rules, tuple matching rules and several functions used in the operational semantics. There are two types of structural congruence rules. The first are for processes and are similar in concept to the asynchronous π-calculus rules. The second type is used for configurations. (SC8) models the commutativity of agents, its purpose being to move to the leftmost position the agent that has to be evaluated. (SC9) moves a new name into the set of global unique names, and (SC10) makes use of the process congruence rules.

3.2. *Restrictions*

The operational semantics presented above makes certain simplifications to the actual $\text{Lime}_{\text{spec}}$. It is our belief that their inclusion would not add value. For completeness we detail them here.

Reactions In Lime_{imp}, reactions are restricted in two ways that are not modelled in our semantics. These restrictions try to prevent the evaluation of reactions from blocking an entire federation. Lime forbids reaction bodies from using blocking operations such as **in** and scopes reactions to the current host of their originating agent. The name *strong reaction* is used to denote such scoped local reactions.

Furthermore, Lime_{imp} provides a kind of distributed reaction, the so-called *weak reactions*, which are implemented by loosening the atomicity of reactions and introducing an asynchronous step between the identification of the candidate tuples and execution of the reactions. Weak reactions are implemented by registering a strong reaction on every host in the federation. These strong reactions will notify the host that registered a weak reaction of the insertion of a matching tuple. Thus, for instance, a rough translation of the remote reaction

$$a_h[\,\mathbf{react}\,\langle\star\star\star s\rangle, x.P \mid Q\,] \mid a'_{h'}[\,Q'\,] \mid a''_{h''}[\,Q''\,]\ \ \mathcal{N}$$

into a set of local reactions is

$$a_h[\,Q\,] \mid a'_{h'}[\,Q'\,] \mid a''_{h''}[\,Q''\,]\ \ \mathcal{N}_T \cup T\mathcal{N}_X$$

where T is the following set of reactions[†]:

$$\langle\langle\star\, a\star s\rangle\,\langle a\, h\rangle\,\langle x\, P\rangle\rangle,$$
$$\langle\langle\star\, a'\star s\rangle\,\langle a'\, h'\rangle\,\langle x\,\mathbf{out}\,\langle x\downarrow_0\ a\, s\rangle\rangle\rangle,$$
$$\langle\langle\star\, a''\star s\rangle\,\langle a''\, h''\rangle\,\langle x\,\mathbf{out}\,\langle x\downarrow_0\ a\, s\rangle\rangle\rangle.$$

[†] We use the syntax $\langle a_0 \ldots a_n\rangle\downarrow_i = a_i$ for selecting values from a tuple.

The original reaction specified on all agents in space s (the query template is $\langle \star \star \star s \rangle$) is replaced by a reaction that guards the portion of the space associated to a (written $\langle \star\, a \star s \rangle$) and two reactions, one for each agent a' and a'' that guard their respective spaces. If one of the latter reactions is triggered, a tuple will be output in a's space, which in turn will trigger P.

Our semantics does not differentiate between strong and weak reactions and does not prevent reactions from blocking the tuple space. Indeed, if

$$\textbf{react}\, v, x.!\, P$$

is triggered, (R1) will never terminate. We will argue, in Section 4, that even with this distinction and the attached restrictions it is possible to block a tuple space, and by extension an entire federation.

Finally, we have not modelled persistent reactions, which are also called once-per-tuple reactions. Modelling these entails changes to the reaction rules so that they do not consume a reaction after it fires. Difficulties with the semantics of once-per-tuple reactions are described in the following section and motivate their omission.

Non-blocking and group operations Lime_{imp} provides both non-blocking input operations and operations that return groups of tuples. Each non-blocking operation can be modelled by extending the semantics with two new reduction rules, one for the case when a matching tuple is immediately found and returned, and one for the case when there is no such tuple and the operation succeeds without blocking the process or modifying the tuple space. Group operations can be defined easily in our formalism by the addition of a reduction rule returning all matching tuples.

Host engagement and disengagement We chose not to model engagement and disengagement of hosts. Engagement could be modelled by creating a set of new agents running on a fresh host identifier. To model disengagement we would have to add a connectivity map to indicate which hosts are connected. While this is not a particularly critical part of the formal model, the engagement protocol raises thorny implementation issues, which will be discussed next.

4. A critique of Lime

During our evaluation we found several inefficiencies and design defects in both $\text{Lime}_{\text{spec}}$ and Lime_{imp} that have to be addressed if Lime, or a similar model, is to gain widespread acceptance. We categorise these issues in three broad classes: *scalability*, *atomicity* and *security*. Scalability issues pertain to the way a system may scale to large configurations, in particular, support for federated operations has to be carefully considered here. Atomicity problems stem from the strong atomicity and consistency imposed by $\text{Lime}_{\text{spec}}$. Even when weakened in the implementation, those requirements make Lime implementations overly complex, full of potential synchronisation problems and quite inefficient. Our experiments with the current system implementation suggest that these inefficiencies affect all applications, even if they do not perform any remote operations. The last category,

security issues, collects some missing features of the model with regards to security and discusses potential attacks on an implementation.

4.1. *Scalability*

4.1.1. *Federated spaces* Federated spaces are distributed data structures that 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 synchronisation required. The approach chosen by Lime_{imp} is to keep a single copy of every tuple on the same host as its owner agent. Federated input requests, such as

$$a_h[\,\mathbf{in}\,\langle \star \star \star s\rangle\,] \;\mid\; a'_{h'}[\,Q\,] \;\; \mathcal{N},$$

in which an agent a queries space s of all agents, are implemented by multicast over the federation. Depending on the size of the federation, multicast may be an onerous choice, but the $\text{Lime}_{\text{spec}}$ is quite clear in that if there is a matching tuple somewhere this tuple should be returned. Thus the entire federation must be searched one way or another.

Blocking remote requests are implemented by the weak reactions described in Section 3.2, which register a strong 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, and 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.

4.2. *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 (Busi and Zavattaro 2001). Consider the configuration

$$a_h[\,\mathbf{move}\,h' \mid \mathbf{move}\,h \mid P\,] \;\; \mathcal{N}_T \oplus \langle v\,a\,a\,s\rangle \mathcal{N}_X.$$

Agent a may move to h' and then return to h – if this does happen, will its tuple $\langle v\,a\,a\,s\rangle$ trigger reactions on h? The answer in our semantics is no. Similarly, $\text{Lime}_{\text{spec}}$ provides an answer to this question by requiring that every tuple be equipped with a globally unique identifier (GUID). 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 reactions may need to store an unbounded amount of data to remember all tuples seen. Unicity of GUIDs can be difficult to ensure in practice. In Lime_{imp}, for instance, agents are moved with Java serialisation. 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. *Atomicity*

4.3.1. *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 a heavy price to pay in a highly concurrent setting. Reaction atomicity implies that the runtime cost of a Lime **out** is entirely unpredictable.

Since reaction bodies are normal programs, termination cannot be guaranteed. In our semantics, the expression $\textbf{react}\, v, x.(!\,\textbf{out}\, v')$ will never terminate as we require that the reaction body reduces to $\mathbf{0}$. $\text{Lime}_{\text{spec}}$ requires that no blocking operations be used in the body of a reaction. But that is hardly enough to guarantee termination. In Lime_{imp} the use of unrestricted Java code fragments in reaction bodies renders their behaviour hard to assess. Once-per-tuple reactions carry an additional risk as they can trigger themselves recursively by outputting a tuple that matches the query template that the reaction is interested in, as in the program

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

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 Lime_{imp}. First, they block the tuple space of the current host, and since disengagement is global and atomic in Lime, they can prevent disengagement procedures from terminating, thus blocking the entire federation.

4.3.2. *Engagement and disengagement* 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 serialised and that other tuple operations are blocked while they are being added to the configuration.

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 using a distributed transaction. 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, loses connectivity, or just crashes, it is highly unlikely that it will have the time to send a message.

4.3.3. *Migration* 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:

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

This is because, regardless of scheduling, the **inp** will always be run in an environment in which 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 period of time during which a transits between hosts. Thus, in Lime_{imp}, the **inp** in the above program can return empty handed, while in $\text{Lime}_{\text{spec}}$ the operation is guaranteed to succeed. A simple way to model this behaviour in the formalisation is to translate every move into a two-step operation, the agent first moves to a distinguished host that is disconnected, in the Lime sense, from every other host, and then, in a second step, moves to its destination.

4.3.4. *Remote input* 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 then is 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 the agent is allowed to move, the system must be ready to handle the additional complexity of messages sent from b's host while a is in transit. In practice, setting up a reliable and efficient message forwarding infrastructure is not trivial.

4.4. *Security*

Provisions for security are minimal. Lime's approach to security is to offer private tuple spaces, which are not merged with other tuple spaces, and are not accessible to other applications. Security breaches may occur as soon as tuple spaces are made public. Communication is in no way protected from malicious applications. Issues such as confidentiality, integrity and accessibility have not found a place in the model. Any application can accidentally or deliberately remove or read a tuple used by other applications to communicate. Such an application can also modify the content of the tuple or even send it multiple times. Nothing prevents applications from trying to remove all tuples or fill up the memory of a host, thus effectively denying service to all co-located agents.

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, which we propose as a basis for building more robust Lime implementations.

5. The CoreLime coordination language

The initial goal of our research was to add security primitives to Lime, but the problems we discovered while trying to understand its implementation convinced us that we had to simplify the model. The root of many of those problems lies in the transparency and atomicity of Lime operations. We have chosen to define a simpler incarnation of Lime, which we call *CoreLime*. This is a non-distributed variant of the basic Lime operation with, as sole distributed operation, agent mobility. The syntax and semantics of most Lime operations is retained with the exception of two important differences. The first is that operations are scoped over the local host only. The second difference relaxes the constraint of atomic execution of reactions at the host level, by allowing reaction handlers to execute in parallel with the other agents.

In CoreLime there is no direct means to operate over remote tuple spaces, nor is there any notion of federation. This choice is motivated by the difficulties we have outlined above. As we were not able to find a semantics of Lime that would yield an efficient and scalable implementation, we chose to retain the main innovation of the original model, namely transient sharing. Furthermore, mobility of agents is theoretically sufficient to encode distributed operations. Though in practice, it is still unclear what is a suitable semantics.

We have retained reactions as they provide a convenient way to monitor the tuple spaces. The main difference with Lime reactions is that CoreLime reactions are run as parallel processes that are spawned whenever a matching tuple is inserted in the tuple space. Since they are not atomic, we can dispense with the restrictions on allowed operations.

The representation of tuples is the same as in $\text{Lime}_{\text{spec}}$: tuples are made up of a value, the names of their owner and destination, as well as the target tuple space. In CoreLime tuple spaces are shared between co-located agents, and when an agent moves to another host, all of its tuples migrate with it.

5.1. *Semantics of CoreLime*

The semantics of CoreLime is presented in Table 4. The one step reduction relation $\rightarrow$ defines evaluation of CoreLime configurations. A configuration $A\ \mathcal{N}$ is composed of an agent expression and an environment. The environment is, as before, a set of tuples $\mathcal{N}_T$ and a set of global names $\mathcal{N}_X$. The definitions of structural congruence, pattern matching and auxiliary functions of Table 3 and Table 2 carry over unchanged. We remind the reader that we use $\mathscr{E}[\![P]\!]$ in place of $a_h[\,P\,] \mid A$.

The input rule (T1) is identical to the corresponding $\text{Lime}_{\text{spec}}$ rule modulo the added side condition that the tuple to be retrieved must reside on the same host as the agent performing the operation, which is written $loc\ v' = h$. The auxiliary function loc is overloaded to yield the location of its argument, which can be either an agent or a tuple. In the case of a tuple, $loc\ v$ returns the location of the agent owning the tuple.

Table 4. *Semantics of CoreLime.*

Reductions

$$\mathscr{E}[\![\mathbf{in}\, v, x\,.\, P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P\{v'/x\} \mid P']\!]\ \mathscr{N}' \quad \text{(T1)}$$

$$\mathscr{E}[\![\mathbf{out}\, v \mid P]\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P]\!] \mid A'\ \mathscr{N}' \quad \text{(T2)}$$

$$\mathscr{E}[\![\mathbf{react}\ v, x\,.\, P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P']\!]\ \mathscr{N}' \quad \text{(T3)}$$

$$a_h[\mathbf{move}\ h'\,.\, P \mid P'] \mid A\ \mathscr{N} \rightarrow a_{h'}[P \mid P'] \mid A \mid A'\ \mathscr{N} \quad \text{(T4)}$$

$$\frac{A\ \mathscr{N} \equiv A'\ \mathscr{N}' \quad A'\ \mathscr{N}' \rightarrow A''\ \mathscr{N}''}{A\ \mathscr{N} \rightarrow A''\ \mathscr{N}''} \quad \text{(T5)}$$

The rules are subjected to the following side conditions:

(T1) $\mathscr{N} = \mathscr{N}'_T \oplus v'\mathscr{N}_X \wedge v \leqslant v' \wedge loc\ v' = h$

(T2) $\mathscr{N}' = \mathscr{N}_T \oplus v'\mathscr{N}_X \oplus r \wedge v' = mkt\ v\, a\, h \wedge A' = react\ r\, h\, \mathscr{N}\, \{v'\} \wedge r \notin \mathscr{N}_X$

(T3) $\mathscr{N}' = \mathscr{N}_T \oplus \langle v\ \langle a\ h\rangle\ \mathbf{in}\, v, x.P\rangle\ \mathscr{N}_X$

(T4) $\mathscr{N}' = (mvt\ \ a\, h'\, \mathscr{N}_T)\, \mathscr{N}_X \oplus r \wedge A' = react\ r\, h'\, \mathscr{N}'\, (sel\ a\, \mathscr{N}) \wedge r \notin \mathscr{N}_X$

Functions

$$\begin{array}{l}
mkt\ \ \langle v\ a'\ s\rangle\, a\, h\ = \langle v\ a'\ a'\ s\rangle, \text{ if } loc\ a' = h \\
mkt\ \ \langle v\ a'\ s\rangle\, a\, h\ = \langle v\ a\ a'\ s\rangle, \quad \text{otherwise}
\end{array}$$

$$\begin{array}{l}
react\ \ r\, h\, \mathscr{N}\ \ = (fold\ (react'\ r\, h\, \mathscr{N}) \mid \mathbf{0}) \\
react'\ \ r\, h\, \mathscr{N}\, v\ = r_h[selr(v\, h\, \mathscr{N}_T)]
\end{array}$$

$$\begin{array}{lll}
selr\ \ v\, h & = (fold\ (selr'\ v\, h) \mid \mathbf{0}) & \\
selr'\ \ v\, h\, \langle v'\, \langle a\, h\rangle\ \mathbf{in}\, v, x.P\rangle & = P\{v/x\}, & \text{if } v' \leqslant v \\
selr'\ \ v\, h\, v' & = \mathbf{0}, & \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
mvt\ \ a\, h & = (fold\ (mvt'\ a\, h) \cup \{\}) & \\
mvt'\ \ a\, h\, \langle v\ a\ a'\ s\rangle & = \langle v\ a'\ a'\ s\rangle, & \text{if } loc\ a' = h \\
mvt'\ \ a\, h\, \langle v\ a'\ a\ s\rangle & = \langle v\ a\ a\ s\rangle, & \text{if } loc\ a' = h \\
mvt'\ \ a\, h\, \langle v\ \langle a\ h'\rangle\ \mathbf{in}\, v, x.P\rangle & = \langle v\ \langle a\ h\rangle\ \mathbf{in}\, v, x.P\rangle & \\
mvt'\ \ a\, h\, v & = v, & \text{otherwise}
\end{array}$$

$$sel\ \ a\, \mathscr{N} = \{v \in \mathscr{N}_T \mid \langle \star\, a\, \star\, \star\rangle \leqslant v\}$$

The output rule (T2) differs from Lime's **out** in that it may spawn a number of reaction processes. The rule is

$$\mathscr{E}[\![\mathbf{out}\, v \mid P]\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![P]\!] \mid A'\ \mathscr{N}'.$$

The new agent term A' is the parallel composition of all reactions that matched the tuple v. Each reaction is run within the body of an anonymous agent, that is, an agent with a fresh name. The side condition for T2 extends the environment with the tuple v' constructed as in $\text{Lime}_{\text{spec}}$ and with a new name r that was not previously in $\mathscr{N}_X$. Finally, all reactions matching v are extracted from the environment and run in parallel.

The reaction rule (T3) is identical to the corresponding rule in $\text{Lime}_{\text{spec}}$. It simply extends the environment with a new reaction. The handler P is protected by an input prefix. Note that the main role of this prefix is to bind x in P: without it x would be a free name.

The CoreLime **move** rule (T4) does the following: it changes the agent's location and relocates its tuples and reactions; delivers all misplaced tuples; and starts all reactions triggered by the moved tuples:

$$a_h[\,\mathbf{move}\ h'\,.\,P \mid P'\,] \mid A \quad \mathcal{N} \quad \rightarrow \quad a_{h'}[\,P \mid P'\,] \mid A \mid A' \quad \mathcal{N}'.$$

Again, the agent term A' represents the agents assigned to execute the reactions triggered by the move.

The auxiliary functions defined for CoreLime's semantics are mostly straightforward. The expression *sel a* $\mathcal{N}$ yields the multiset of tuples owned by *a*. When applied to a tuple v, a location h and a multiset $\mathcal{N}_T$, the *selr* function selects matching reactions and returns the parallel composition of their handler processes. If none is found, **0** is returned. *selr* applies *selr′* to each element in turn. This latter function either returns the inert process or, if a matching reaction is found, returns the body of the handler with the matched tuple substituted in $P\{v/x\}$. The expression *react′ r h* $\mathcal{N}$ *v* yields an agent *r* with, as body, the parallel composition of all reactions matching v. The expression (*react r h* $\mathcal{N}$) T function yields the parallel composition of a set of agents, one agent per tuple in the multiset T. The body of each of these agents is generated by *react′*.

5.2. *CoreLime capabilities*

As we mentioned earlier, Lime has no provisions for security, making it particularly exposed to a variety of attacks. We extend CoreLime with a fine-grained access control mechanism based on the notion of capabilities. Capabilities can be construed as tokens to be presented by a principal to gain access to a resource. In CoreLime, agents are the principals and tuple spaces are the resources. Thus capabilities regulate tuple space operations. Every tuple space operation is controlled by a capability, which can be granted to one or several agents. In our formalisation we choose to represent capabilities by the following global names: **cr** denotes the creator of a tuple space; **in** allows the reading of a tuple space; **out** allows the writing to a space; and **react** allows the registering of a reaction. Capabilities are stored in the environment as tuples with the following fields: capability type *r* (either , **cr**, **in**, **out** or **react**); an owner *a*; a destination a'; and a tuple space name *s*. The destination a' is the agent that will be granted the capability *r*. The owner *a* is either the agent who issued the capability or the destination. The tuple space *s* is the name of the space for which the capability applies.

The operational semantics of the capabilities is presented in Table 5 as an addition to CoreLime's semantics, and the syntax of the calculus is extended in the obvious way. We only present new rules and side conditions.

The rule (TN) shows how to construct a new tuple space in the extended calculus:

$$\mathscr{E}[\![\mathbf{nts}\ s.P \mid P']\!] \quad \mathcal{N} \quad \rightarrow \quad \mathscr{E}[\![((v\ s)\ P \mid \mathbf{out}\ \langle \mathbf{cr}\ a\ s\rangle \mid \mathbf{out}\ \langle \mathbf{cr}\ a\ s\rangle) \mid P']\!] \quad \mathcal{N}'.$$

Table 5. *Semantics of Capabilities.*

Reductions

$$\mathscr{E}[\![\mathbf{nts}\ s.P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![((\nu\, s)\, P \mid \mathbf{out}\ \langle \mathbf{cr}\ a\ s\rangle \mid \mathbf{out}\ \langle \mathbf{out}\ a\ s\rangle) \mid P']\!]\ \mathscr{N}' \quad \text{(TN)}$$

$$\mathscr{E}[\![\mathbf{cap}\ \langle r\ a'\ s\rangle\ .P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![\mathbf{out}\ \langle r\ a'\ s\rangle \mid P \mid P']\!]\ \mathscr{N}' \quad \text{(T0)}$$

The rules, including the corresponding ones from Table 4 are subjected to the following side conditions:

(T0) $\exists\langle \mathbf{cr}\ a\ a\ s\rangle \in \mathscr{N}_T$ (T2) $\exists\langle \mathbf{out}\ a\ a\ s\rangle \in \mathscr{N}_T$ (T3) $\exists\langle \mathbf{react}\ a\ a\ s\rangle \in \mathscr{N}_T$

(T1) $\exists\langle \mathbf{in}\ a\ a\ s\rangle \in \mathscr{N}_T \wedge v \notin \{\mathbf{cr}, \mathbf{in}, \mathbf{out}, \mathbf{react}\}$

The operation creates a fresh name s and registers the agent a as the creator of that space. For every space there is a single creator that is initially allowed to output to the space and to grant capabilities.

A capability is issued using rule (T0). The operation takes three arguments, the capability, a principal and a tuple space:

$$\mathscr{E}[\![\mathbf{cap}\ \langle r\ a'\ s\rangle.P \mid P']\!]\ \mathscr{N} \rightarrow \mathscr{E}[\![\mathbf{out}\ \langle r\ a'\ s\rangle \mid P \mid P']\!]\ \mathscr{N}'.$$

The side condition of (T0) checks that the operation is issued by the creator of the tuple space. Capabilities, like all tuples, belong to agents, are stored in their share of the tuple space and are relocated whenever their owner moves.

The side conditions (T1–3) check the existence of a capability allowing the agent to perform the desired operation on the specified tuple space. Note that originally the creator of a tuple space does not have the right to read or register reactions and must explicitly enable those rights. Furthermore, (T1) must prevent agents from removing capabilities from the tuple space.

As an example of the use of capabilities we show how an agent sends its public key to a co-located agent:

$$a_h[\,\mathbf{nts}\ s.\mathbf{cap}\ \langle \mathbf{in}\ b\ s\rangle\ .\mathbf{out}\ \langle k\ b\ s\rangle\,] \mid b_h[\,\mathbf{inp}\ v, x\langle \star\ b\ b\ s\rangle\,].$$

Agent a creates a new tuple space s and grants agent b the right to read from it. It then sends its public key k to b in the newly created tuple space. Agent b just reads from tuple space s. This assumes that both agents know the tuple space name s *a priori*. Even if another agent knows s, it will not be capable of accessing it, as the only capability was issued for b.

In this simple capability model we have chosen to disallow capability transfer, that is, only the owner of a tuple space is permitted to grant capabilities to that space. Since capabilities are first class entities, a weaker security model in which capability can be exchanged could easily be modelled.

5.3. *Implementation details*

The current implementation of CoreLime is written in Java, and is built on top of the TSpaces (Lehman *et al.* 1999) implementation of tuple spaces. A platform on which agents execute runs on every device belonging to the federation and listens on a default port for incoming agents. Every platform has a host-level public tuple space, uniquely determined by the host address and platform's port number, and every agent running on the platform is given a reference to it. The purpose of this host tuple space is to allow co-located agents to exchange information, such as capabilities. Following Lime's requirements, we also provide a host-level *system tuple space*, which is used to store system information such as what applications are present at any time.

As specified in CoreLime's semantics, agents can access the tuple spaces of other agents only if located on the same host. Agents can move between hosts, and their share of tuples and reactions migrates with them and is merged on the destination host. An agent moves by specifying a file containing its code, while its state is saved using serialisation. When receiving the code and state of an agent, a platform loads the class files with a local class loader and deserialises the state using the classes already loaded.

Readers interested in a further description of the CoreLime interface as well as in experimental results are referred to Carbunar *et al.* (2001).

6. Related work

The operational semantics presented in this paper resembles the ambient calculus of Cardelli and Gordon (Cardelli and Gordon 1998). In the ambient calculus, ambients containing a set of running processes can enter and exit ambients. Processes running in an ambient communicate by exchanging asynchronous messages, but the primitive used for reading messages is not based on pattern matching, so communicating processes must know each other's identity. Also, processes cannot transparently read messages located in sibling ambients. Like the ambient calculus, CoreLime relies on migration to support remote communication.

Busi and Zavattaro have also proposed a formalisation of transiently shared tuple spaces (Busi and Zavattaro 2001). They model local versions of **in** and **inp** but they do not model reactions and do not consider the impact of supporting federating tuple spaces in a real system.

KLAIM (de Nicola *et al.* 1998) provides a coordination language based on Linda's primitive operations. It's main idea is the notion of *explicit localities*, which is an abstraction mechanism over hosts that allows tuples and processes to be sent and retrieved from remote hosts. KLAIM is also extended with a type system that statically enforces security properties, such as checking of the access rights.

TuCSoN (Omicini and Zambonelli 1999) 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 inter-agent communication and to access local resources. Tuple spaces have unique names at the host level, and Linda like operations can be performed remotely

on them by specifying their name and the name of the host. In addition, TuCSoN extends tuple spaces with the notion of behaviour specification, which are similar to reactions.

The concept underlying reactive tuple spaces is very close to the publish/subscribe paradigm (Carzaniga *et al.* 2001; Eugster *et al.* 2000) if we view the data structures representing events as tuples. In addition, the publish/subscribe model provides flow decoupling of interlocutors, as neither publisher nor subscriber waits for the operation to succeed. As future work, we could build efficient and scalable remote operations using an implementation of a publish/subscribe system. A remote **in** can be viewed as a blocking subscribe to the occurrence of a tuple matching a specified template, and an **out** can be a publish of an event containing the argument tuple.

7. Conclusion

This paper has provided a semantics of the Lime middleware for mobile environments. This infrastructure is being proposed as a communication model for mobile systems. Our investigation has uncovered a number of deficiencies in the specification and implementation of Lime. Our reaction was to propose and implement a simplified system called CoreLime. CoreLime retains the key feature of its predecessor, namely transient sharing of tuple spaces, but does away with distributed operations and loosens the original system's stringent atomicity requirements.

We view this paper as a starting point for a number of research directions. One important question is whether a sensible semantics for remote operations can be defined and implemented. Another open problem is related to efficiency, since in our current implementation some of the query operations can be quite costly, and much more so than message passing through named channels. Finally, type and effect systems for CoreLime are needed in order to build larger distributed applications out of numerous cooperating agents.

References

Bryce, C., Oriol, M. and Vitek, J. (1999) A Coordination Model for Agents Based on Secure Spaces. In: Ciancarini, P. and Wolf, A. (eds.) Proc. 3rd Int. Conf. on Coordination Models and Languages. *Springer-Verlag Lecture Notes in Computer Science* **1594** 4–20.

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

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

Busi, N. and Zavattaro, G. (2001) Some Thoughts on Transiently Shared Tuple Spaces. *Workshop on Software Engineering and Mobility. Co-located with International Conference on Software Engineering*.

Carbunar, B., Valente, M. T. and Vitek, J. (2001) Lime revisited. Proceedings of the Fifth IEEE Conference on Mobile Agents. *Springer-Verlag Lecture Notes in Computer Science* **2240**.

Cardelli, L. and Gordon, A. (1998) Mobile Ambients. In: Nivat, M. (ed.) Foundations of Software Science and Computational Structures. *Springer-Verlag Lecture Notes in Computer Science* **1378** 140–155.

Carzaniga, A., Rosenblum, D. S. and Wolf, A. L. (2001) Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* **19** (3) 332–383.

de Nicola, R., Ferrari, G. L. and Pugliese, R. (1998) Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering* (*Special Issue on Mobility and Network Aware Computing*).

DeNicola, R. and Pugliese, R. (1996) A Process Algebra based on Linda. In: Ciancarini, P. and Hankin, C. (eds.) Proc. 1st Int. Conf. on Coordination Models and Languages. *Springer-Verlag Lecture Notes in Computer Science* **1061** 160–178.

Eugster, P. T., Guerraoui, R. and Sventek, J. (2000) Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In: Bertino, E. (ed.) ECOOP 2000 – Object-Oriented Programming. *Springer-Verlag Lecture Notes in Computer Science* **1850** 252–276.

Fournet, C. and Gonthier, G. (1996) The reflexive chemical abstract machine and the join-calculus. In: *Proceedings of POPL'96*, ACM 372–385.

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

Lehman, T. J., McLaughry, S. W. and Wycko, P. (1999) T spaces: The next wave. In: *HICSS*.

Milner, R., Parrow, J. and Walker, D. (1992) A calculus of mobile processes, Parts I and II. *Journal of Information and Computation* **100** 1–77.

Morin, J.-H. (1998) HyperNews: a Hyper–Media Electronic Newspaper based on Agents. In: *Proceedings of HICSS-31, Hawaii International Conference on System Sciences*, Kona, Hawaii 58–67.

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

Murphy, A. T. (2000) *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*, Ph.D. thesis, Washington University, St. Louis.

Omicini, A. and Zambonelli, F. (1999) Tuple Centres for the Coordination of Internet Agents. In: *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, ACM 183–190.

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

Vitek, J. (1999) *The Seal model of Mobile Computations*, Ph.D. thesis, University of Geneva.

Wojciechowski, P. T. and Sewell, P. (2000) Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency* **8** (2) 42–52.