

Making Tuple Spaces Safe for Heterogeneous Distributed Systems

Naftaly H. Minsky
Department of Computer
Science
Rutgers University
minsky@cs.rutgers.edu

Yaron M. Minsky
Department of Computer
Science
Cornell University
yminsky@cs.cornell.edu

Victoria Ungureanu
Department of MSIS
Rutgers University
ungurean@rbs.rutgers.edu

ABSTRACT

Linda is a high level communication model which allows agents to communicate via a shared tuple spaces without knowing each other's identities and without having to arrange for a definite rendezvous. This high level of abstraction would make Linda particularly suitable for use as a coordination model for heterogeneous distributed systems, if it were not for the fact that the Linda communication is unsafe.

In order to enhance the safety of tuple spaces, this paper introduces a mechanism for establishing security policies that regulate agent access to tuple spaces. Our mechanism is based on a previously published concept of law-governed interaction. It makes a strict separation between the formal statement of a policy, which we call a "law," and the enforcement of this law, which is carried out by a set of policy-independent trusted *controllers*. A new policy under this scheme is created basically by formulating its law, and can be easily deployed throughout a distributed system.

Two examples policies are discussed here in detail: one ensures a secure bidding policy; the other prevents denial of service, by regulating the flow of requests sent to the tuple spaces.

1. INTRODUCTION

In the mid-80's, the Linda Project [3] was created to simplify communication and synchronization for parallel programs. Although Linda was originally intended for use with tightly integrated systems, there has been increasing interest in applying the Linda communication model (and in particular, the notion of a tuple space) to the task of building middleware for heterogeneous distributed systems, or *open* systems.

Interest in the use of tuple space like abstractions for open systems has gained attention recently with the introduction of systems like Sun's *JavaSpaces* [8] and IBM's *TSpaces* [16], which marry a Linda-like tuple space to the Java program-

ming language. These systems are envisioned as a kind of universal "network dialtone", a communications fabric that a wide array of divergent systems can use to communicate with each other. The intended range of applications is quite large, from allowing for configuration and mutual discovery of different pieces of hardware installed on a LAN, to coordinating the provision of services such as local restaurant listings and remote repair diagnostics to travelers in their cars [10].

There is much to be said for the use of Linda as a coordination model for open systems [9]. In particular, Linda uncouples communicating agents in both time and space by allowing agents to communicate without knowing each other's identities and without having to arrange for a definite rendezvous.

Unfortunately, Linda's reliance on shared, wide open tuple spaces makes it unsafe for use in open systems. This is clearly a security concern: a malicious client with access to a given tuple space could disrupt any system that depends on the integrity of the data stored in that tuple space. But even when security is not an issue i.e., when all clients of a given tuple space are assumed to be non-malicious, the use of unprotected tuple spaces is still a threat to system stability. A buggy agent could easily corrupt a shared tuple space, thus disrupting the activities of the other clients of the tuple space. This could lead to the kind of ugly scenario where, say, adding a VCR to your home network could cause your garage-door opener to stop working.

This deficiency of Linda has been noted before. Ciancarini [4], among many others, enhanced Linda by making it support a sophisticated multiple tuple space organization. Both *JavaSpaces* and *TSpaces* have adopted the use of multiple tuple spaces, and provide simple access control on a per tuple space basis.

Unfortunately, segregating communication into multiple tuple spaces increases safety only insofar as it eliminates sharing. But tuple spaces are most useful when diverse agents share access to a single tuple space. Moreover, the content-based nature of retrieval from tuple spaces requires a content-based access control. As we shall see, control expressed purely in terms of access to named subspaces is not powerful enough to establish many useful regimes, such as safe peer-to-peer communication (i.e., message exchange) via a tuple space.

To provide for such content-based access control, a new model for Linda, called law-governed Linda, has been proposed in [11], but had never been fully implemented. In

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright ACM 0-89791-88-6/97/05 ..\$5.00

this paper we show that a similar, but more powerful, control mechanism can be established without changing the Linda model itself¹, by subjecting the interaction of tuple spaces with their clients to a coordination regime called law-governed interaction (LGI). As we shall see, LGI can be used to establish a wide range of protocols for coordination via tuple spaces, including secure peer-to-peer communication. We start in Section 2 with a motivating example, making the case for content-based control over access to tuple spaces. In Section 3 we provide a summary of LGI, and explain briefly how it is applied to tuple spaces. In Section 4 we discuss two examples. The theoretical efficiency of the mechanism, and the performance of its current implementation, via the Moses toolkit, are discussed in Section 5; Section 6 discusses some related work, and we conclude in Section 7.

2. MESSAGE-PASSING—A MOTIVATING EXAMPLE

To illustrate the weaknesses of conventional Linda we will demonstrate that this coordination model cannot support even a simple form of pairwise communication (i.e., message passing) which is secure from eavesdropping, stealing and from forging. We employ here, and in the rest of this paper, a Prolog implementation of Linda, provided by BinProlog [14]. We start with some comments about the syntax of this Linda implementation:

A tuple under this Linda is a list of prolog terms, such as

```
[person,name(jones),age(23)],
```

which may be used to represent a person with the specified name and age. More generally, a field of a tuple is represented by a term $t(v)$, where t is a symbol that specifies the *type* of the field, and the possibly empty v represents its *value*—which in most of our examples would be a literal, but could be a general Prolog-term. The Linda concept of a “formal” component of a template is realized here by a variable, represented by a capitalized symbol. And the matching of a template to tuples is defined by unification.

Now, suppose that a pair of agents interacting via a tuple space needs to exchange messages between them; since they may not even have the IP-address of each other, they would like to do this exchange via the tuple space. It might seem that such exchange of messages can be easily accomplished under Linda simply by adopting the convention that a tuple

```
[msg(m),from(s),to(t)]
```

represents a message m in transit from agent s to agent t —that is, that only agent s **out**’s such a “message-tuple,” and that only agent t **in**’s it.

Unfortunately, such realization of message passing would be unsafe, because it relies on a *voluntary convention* that needs to be followed by all processors accessing the tuple space—not just the two who are communicating. There are two ways in which this convention can be violated.

1. Any agent interacting with the tuple space in question can read, and even remove, message tuples that, by our convention, are intended for a given agent t .

2. Any agent can **out** message tuples that, by our convention, appear to have been sent by some agent s , thus effectively *forging* a message from s .

One can try to provide for message exchange by means of multiple-tuple spaces, as follows: For agent s to send a message to t , it inserts an appropriate message-tuple into a subspace that can be accessed only by t and s . This provides us with (nearly) the guarantee that we are looking for, but unfortunately it requires that there is a subspace with the appropriate access control settings for every pair of agents. Setting up such quadratic number of subspaces for every pair of agents would be time-consuming at best, and requires a special mechanism, for such subspaces to be created dynamically and automatically. As we shall see, this problem becomes simple given the ability to impose *content sensitive* constraints on Linda-operations.

The difficulty of using Linda for secure message passing has been noted by Pinakis [13], who constructed a special variant of Linda that supports such message passing. We, on the other hand, can provide secure message passing as one of many types of policies expressible under LGI, and without changing the Linda model itself.

3. LAW-GOVERNED INTERACTION (LGI)

This section is a summary and adaptation of the description of LGI in [12]. This mode of interaction is currently supported by a toolkit called Moses, which is implemented mostly in Java². The organization of this section is outlined after we introduce our concept of coordination policy, or “policy,” for short.

3.1 The Concept of a Coordination Policy

DEFINITION 1. A *coordination policy* \mathcal{P} is a four-tuple $(\mathcal{M}, \mathcal{G}, \mathcal{CS}, \mathcal{L})$ where

1. \mathcal{M} is the set of messages regulated by this policy—they are called \mathcal{P} -messages. (In this paper we will be concerned only with messages exchanged between a tuple space and its clients.)
2. \mathcal{G} is an open group of agents that exchange \mathcal{P} -messages subject to law \mathcal{L} —it is called the “policy-group” of \mathcal{P} . Group \mathcal{G} is open in two orthogonal respects:

- (a) \mathcal{G} is heterogeneous.
- (b) The membership in \mathcal{G} can change, subject to the law \mathcal{L} of this policy.

In the context of this paper \mathcal{G} would include at least one *Linda-server*, representing a tuple space, and its *clients*.

3. \mathcal{CS} is a mutable set $\{\mathcal{CS}_x \mid x \text{ in } \mathcal{G}\}$ of what we call control states, one per member of group \mathcal{G} .
4. \mathcal{L} represents the “rules of engagements” between the members of group \mathcal{G} , formulated in such a way that it can be enforced locally, at each member. The law regulates: (a) the exchange of messages between each member of \mathcal{G} and the rest of this group, and (b) the effect of this exchange on the control-state of each member.

¹The model described in [11] involved a substantial change in the Linda mechanism

²One module of Moses is still written in Prolog.

The rest of this section is organized as follows: the structure and the interpretation of laws is discussed in Section 3.2, and the law-enforcement mechanism currently employed by the Moses toolkit is discussed in Section 3.3.

3.2 On the Structure and Interpretation of Laws

Abstractly speaking, the law \mathcal{L} of a policy is a *function* that returns a ruling for every possible regulated-event that might happen at a given agent. The ruling returned by the law is a possibly empty sequence of primitive operations, which is to be carried out locally at the location of the event from which the ruling was derived (called the *home* of the event). (By default, an empty ruling implies that the event in question has no consequences—such an event is effectively ignored.)

Concretely, under the Moses implementation of LGI, the law is defined by means of a Prolog-like program³ L which, when presented with a goal e , representing a regulated-event at a given agent x , is evaluated in the context of the control-state of this agent, producing the list of primitive-operations representing the ruling of the law for this event. Before introducing the structure of such laws we define the types of events that are regulated by laws under Moses, the structure of an agent’s control-state, and the primitive operations that can be included in the ruling of a law.

The Regulated Events: The events that are subject to the law of a policy are called *regulated events*. Each of these events occurs at a certain agent, called the *home* of the event. The following are two of these event-types.

1. **sent(x, m, y)**—occurs when agent x sends an \mathcal{L} -message m addressed to y . The sender x is considered the *home* of this event.
2. **arrived(x, m, y)**—occurs when an \mathcal{L} -message m sent by x arrives at y . The receiver y is considered the *home* of this event.

The Control State: The *control-state* CS_x of a given agent x is the bag of attributes associated with this agent (represented here as Prolog terms). These attributes are used to structure the group \mathcal{G} , and provide state information about individual agents, allowing the law \mathcal{L} to make distinctions between different members of the group. The control-state CS_x can be acted on by the primitive operations, which are described below, subject to law \mathcal{L} . Note that, for the purpose of this paper, laws, unlike a control-states, are assumed to be *immutable*.

The Primitive Operations: The operations that can be included in the ruling of the law for a given regulated event e , to be carried out at the home of this event, are called *primitive operations*. It is only through these primitive operations that a regulated event can have any effect on an agent or an agent’s control state. These operations include:

1. **Operations on the control-state:** These operations update the control-state of the home agent. They include: (1) $+t$ which adds the term t to the control state; (2) $-t$ which removes a term t ; (3) $t1 \leftarrow t2$ which

replaces term $t1$ with term $t2$; (4) $incr(t(v), d)$ which increments the value of the parameter v of a term t with quantity d (v and d are assumed here to be integers.); and (5) $dcr(t(v), d)$ which decrements the value v of a term t with some quantity d .

2. Operations on messages:

- Operation **forward(x, m, y)** sends message m to y , where x identifies the sender of the message. The receipt of the message will trigger at y an **arrived(x, m, y)** event.
- Operation **deliver(x, m, y)** delivers the message m to the home-agent y , where x is the nominal sender of this message. Unlike **forward**, the receipt of this message at y does not trigger any event.

The Law: The program L representing a law is written in a restricted version of Prolog that, in particular, does not permit asserts and calls. In addition to the standard types of Prolog goals the body of a rule may contain two distinguished types of goals that have special roles to play in the interpretation of the law. These are the *sensor-goals*, which allow the law to “sense” the control-state of the home agent, and the *do-goals* that contribute to the ruling of the law. A *sensor-goal* has the form $t@CS$, where t is any Prolog term. It attempts to unify t with each term in the control-state of the home agent. A *do-goal* has the form $do(p)$, where p is one of the above mentioned primitive-operations. It appends the term p to the ruling of the law.

3.3 The Distributed Law-Enforcement Mechanism

The most critical aspect of our concept of a policy is the assumption that its law is observed by all members of the policy-group \mathcal{G} . Given the openness and heterogeneity of \mathcal{G} , this assumption must be supported by strict enforcement of the law. By “enforcement” we do not mean that anybody is coerced to participate in a given policy \mathcal{P} , but that any exchange of \mathcal{P} -messages, once undertaken, satisfies law of \mathcal{P} . We now describe how such enforcement is carried out under Moses.

Law \mathcal{L}_P is enforced by a set of trusted entities called *controllers* that mediate the exchange of \mathcal{P} -messages between members of group \mathcal{G}_P . For every active member x in \mathcal{G}_P , there is a controller C_x logically placed between x and the communications medium, as illustrated in Figure 1. All these controllers have identical copies of law \mathcal{L}_P , and each controller maintains the control-states of the agents under its jurisdiction. In the current implementation this is achieved as follows: a policy \mathcal{P} is maintained by a server that provides persistent storage for the law \mathcal{L} of this policy, and the control-states of its members. This server is called the *secretary* of \mathcal{P} , to be denoted by S_P .

For an agent x to be able to exchange \mathcal{P} -messages under a policy \mathcal{P} , it needs to engage in a connection protocol with the secretary. The purpose of the protocol is to assign x to a controller C_x which is fed the law of \mathcal{P} and the control state of x (for a detailed presentation of this protocol the reader is referred to [12]).

We are in position now to explain how the exchange of \mathcal{L} -messages gets to be mediated by controllers, and how this

³Prolog is incidental to this model, and can, in principle, be replaced by a different, possibly weaker, language; for now, a restricted version of Prolog is being used.

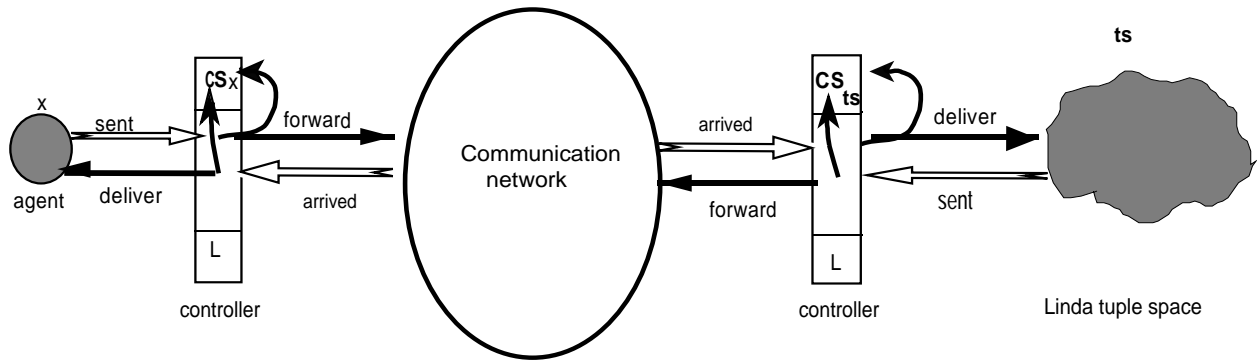


Figure 1: Enforcement of the Law

mediation is carried out. Consider, for example, an agent x sending a \mathcal{P} -message m to a tuple space ts , assuming that both x and ts have joined the policy-group \mathcal{G}_P . (For a discussion of how does one join a policy-group the reader is referred to [12].) Message m is sent by means of a routine provided by the Moses toolkit, which forwards it to C_x —the controller assigned to x . When this message arrives at C_x , it generates a **sent**(x, m, ts) event at it. C_x then evaluates the ruling of law \mathcal{L}_P for this event, taking into account the control-state CS_x that it maintains, and carries out this ruling.

If this ruling calls for message m to be forwarded to ts , then C_x would send m to the controller C_{ts} assigned to ts . And if the ruling calls the control-state CS_x to be updated, such update is carried out directly by C_x .

When the message m sent by C_x arrives at C_{ts} it generates an **arrived**(x, m, ts) event. Controller C_{ts} computes and carries out the ruling of the law for this event. This ruling might, for example, call for m to be delivered to ts , and for the control-state CS_{ts} maintained by C_{ts} to be modified.

In general, all regulated events that occur nominally at an agent x actually occur at its controller C_x . To avoid race conditions, the events pertaining to x are handled *sequentially* in chronological order of their arrival as follows: the controller evaluates the ruling of the law for each event, and carries out this ruling, atomically, so that the sequence of operations that constitute the ruling for one event do not interleave with those of any other event occurring at x .

In general, all regulated events that occur nominally at an agent x actually occur at its controller C_x . The events pertaining to x are handled *sequentially* in chronological order of their occurrence. The controller evaluates the ruling of the law for each event, and carries out this ruling *atomically*, so that the sequence of operations that constitute the ruling for one event do not interleave with those of any other event occurring at x . Note that a controller might be associated with several agents, in which case events pertaining to different agents are evaluated concurrently.

4. EXAMPLES

We present here two quite different examples of LGI-based coordination via tuple spaces. The first example is that of a policy that supports secure interaction between service providers and their customers; this example includes secure message passing of the kind discussed in Section 2. Our second example shows how a tuple space can protect itself from getting congested, by controlling the frequency of messages sent to it by its users. These examples have been tested with Moses toolkit, applied to the T-Space system of IBM and to the BinProlog implementation of Linda.

4.1 A Secure Bidding Policy

Suppose that we want to use a tuple space ts as a medium for communication between the providers of certain services, and their clients. Such communication is to have the following steps: (a) when a client c needs a service s he *outs* into ts a *request-tuple*

$[requester(c), service(s)],$

identifying itself and the service requested; (b) service provider bid for this service by *outing bid-tuples* of the form:

$[offerFor(c, s), fee(f), provider(p), contact(addr)],$

identifying the request the bid is for (via its two arguments c and s), the fee f for the service being offered, the id p of the provider making the bid, and an address of this provider, which the client can use to establish contact with him outside of the tuple space (say, an e-mail address of the provider); (c) finally, the client chooses among the bids he receives for his request, and establishes contact with the chosen provider, via the contact address included in the bid. To safeguard this communication we would like to establish the following constraints:

1. Request-tuples and bid-tuples cannot be forged. That is, each such tuple should correctly identify the agent that made it.

2. A request-tuple issued by a client c can be read by any provider (but not by clients), and can be removed *only* by c himself.
3. A bid issued for a request of client c can be accessed (via *in*) only by c himself. In other words, a bid-tuple made by a provider p for a service request by client c should behave like a secure message (in the sense of Section 2) from p to c .

Such secure bidding is supported under LGI by law \mathcal{L}_{SB} in Figure 2. This figure starts with the description of the initial control-state of various agents⁴. It then lists the rules of this law, each followed by a comment (in italic), which, together with the following discussion, should be understandable even for a reader not well versed in Prolog. Let us examine now this particular law in detail:

<p><i>Initially:</i> Each provider has the term <code>serviceProvider</code> in its control state.</p> <p>R1. <code>sent(C1, out([requester(C2), service(S)]), ts) :- C1==C2, do(forward).</code> <i>Any client C can out request-tuples for services, if identified as being from himself.</i></p> <p>R2. <code>sent(C1, in([requester(C2), service(S)]), ts) :- C1==C2, do(forward).</code> <i>Any client C can in (i.e. read and remove) request-tuples previously posted by himself.</i></p> <p>R3. <code>sent(P, rd([requester(C), service(S)]), ts) :- serviceProvider@CS, do(forward).</code> <i>Any service provider P can read requests posted to the tuple space.</i></p> <p>R4. <code>sent(P1, out([offerFor(C,S), fee(f), provider(P2), contact(Addr)]), ts) :- P1==P2, serviceProvider@CS, do(forward).</code> <i>Any service provider P can out offer-tuples, if identified (in term provider, as being from himself).</i></p> <p>R5. <code>sent(C1, in([offerFor(C2,S), fee(f), provider(P), contact(Addr)]), ts) :- C1==C2, do(forward).</code> <i>Only the client who posted a service request can in a corresponding offer.</i></p> <p>R6. <code>sent(ts, _, _) :- do(forward).</code> <i>Any message sent by ts is forwarded.</i></p> <p>R7. <code>arrived(., ., .) :- do(deliver).</code> <i>when a message arrives anywhere, it is delivered.</i></p>
--

Figure 2: Law \mathcal{L}_{SB} of Secure Bidding

- Rule $\mathcal{R}1$ of this law is the only one that provides for the outing of request-tuples, and it requires the `requester` field of the tuple to be identical with the id of the sender of this message. Thus, this rule ensures that request-tuples cannot be forged, as required in (1) above.

- Rules $\mathcal{R}2$ and $\mathcal{R}3$ provide for the retrieval of request-tuples, imposing constraint (2) above: Rules $\mathcal{R}2$ allows

⁴For a discussion of how the initial control state is established the reader is referred to [12].

a request tuple to be in-ed only by the client that outed it. And Rules $\mathcal{R}3$ allows any request tuple to be read (but not removed) by any producer

- Rule $\mathcal{R}4$ allows providers to issue bid-tuples. It also ensures that the bidder identifies himself correctly in the `provider` term of the bid-tuple, so that such tuples cannot be forged, as required in (1) above.
- By rule Rule $\mathcal{R}5$, an offer issued by any provider for a request by client c , can be in-ed only by c —establishing constraint (3).
- Finally, Rule $\mathcal{R}6$ allows the tuple space ts to send arbitrary messages to any agent—these are replies that ts send to its users. And Rule $\mathcal{R}7$ permits all arriving messages to be delivered, to any recipient. (Note that under this particular law arrivals need not be regulated because all regulation is done when messages are sent. This is not the case in general, as our second example demonstrates.)

4.2 Congestion Control Policy

We now consider the following policy designed to allow a tuple space to protect itself from getting congested, by controlling the frequency of messages sent to it by its users:

1. Every client of a tuple space ts has a quantum of time dt assigned to it, which is to be the *minimal delay* between any two requests sent by this agent to the tuple space. (If an agent attempts to send a message to ts sooner than permitted by its delay, the message is to be blocked.)
2. The server of the tuple space can set the delay of an agent to any desired value.

This policy is established by law \mathcal{L}_{CC} , presented in Figure 3, as follows: By Rule $\mathcal{R}1$ of this law a message sent by a client to the tuple space ts is forwarded only if the delay condition between messages is satisfied. If the sender issues its request too early, the message will not be forwarded and is thus effectively blocked. This rule, then, gives the term `delay(dt)` in the CS of an agent the effect intended for it in our policy.

By Rule $\mathcal{R}2$, messages sent by ts are forwarded directly to their destination. Most of these messages carry replies to clients, and they are delivered directly upon arrival, according to Rule $\mathcal{R}4$. But when a message `changeDelay(val)` sent by ts to a client c , then, by Rule $\mathcal{R}3$, the value of the term `delay` in the control state of c is set to `val`. This is what provides the tuple space with the ability to adjust at will the `delay` term of his clients.

Note that the `changeDelay` message effects the control state of a client, but is not delivered to the client itself, thus preserving the usual semantics that a client only receives replies for its requests. By Rule $\mathcal{R}4$ all other messages sent by the tuple space are delivered to the clients⁵.

Finally, note that for this policy to be effective, it must be enforced at the client side (by the client’s controller, in our

⁵with the exception of operations on `changeDelay` tuples, which are blocked. Allowing clients to issue requests for such tuples would have allowed for the possibility that an agent could change its own delay by performing a `in(changeDelay(x))` followed by a `rd/out(changeDelay(x))`.

Initially: Each client has in its control state: (1) a term `delay(DT)` where `DT` represents the minimum delay between successive messages sent by the client to the tuple space `ts`; and (2) a term `lastCall(Tlast)` where `Tlast` is the time when the last message was sent to the tuple space (initially set to 0).

```

R1. sent(C,M,ts) :-
    lastCall(Tlast)@CS,delay(DT)@CS,clock(T),
    T > (Tlast + DT),
    do(lastCall(Tlast)←lastCall(T)),
    do(forward).

    A message sent to the tuple space ts will be forwarded
    only if the delay condition is satisfied. If the message is
    forwarded, the term lastCall is updated to reflect that a
    message is sent at the current time T.

R2. sent(ts,_,_) :- do(forward).

    Any message sent by the tuple space s is forwarded to its
    intended destination.

R3. arrived(ts,changeDelay(Val),X) :-
    do(delay(DT)←delay(Val)), do(deliver).

    When a message changeDelay(Val) sent by ts arrives at
    the destination, the delay term is changed to Val.

R4. arrived(_,M,_)
    :- not (M=changeDelay(V)),do(deliver).

    Any message other than changeDelay arriving at the des-
    tination is delivered without further ado.

```

Figure 3: Law \mathcal{L}_{CC} - Congestion Control Policy

case). Otherwise, if this policy is checked at the server side, the server might be congested just from checking the validity of messages sent to it, even if most of them end up being rejected.

5. ON THE EFFICIENCY OF LGI, AND OF ITS MOSES IMPLEMENTATION

The current implementation of Moses, which has been tested on Solaris and Windows NT platforms is experimental and much less efficient than it can be—presently, an event is evaluated in approximately 3.5 ms. But even in its present state, LGI is quite affordable, under a wide range of applications.

We start this section by analyzing the structure of the relative overhead incurred when sending a message under LGI; we then evaluate the relative overhead under different scenarios. Finally, in Section 5.2 we report on the testing of the performance (efficiency and robustness) of our current prototype implementation of Moses.

5.1 The Relative Overhead of LGI

Consider a message `m` sent by an agent `x` to a tuple space `ts`. If the interaction between the two parties is mediated by controllers in the manner described in Section 3, then this message would be converted to three consecutive messages: (1) from `x` to C_x , (2) from C_x to C_{ts} , and (3) from C_{ts} to `ts`. The overhead $o_{x,y}$, due to the extra messages and the law-evaluations involved, is given by the following formula:

$$o_{x,y} = (t_{com}^{x,C_x} + t_{eval}^{sent} + t_{com}^{C_x,C_{ts}} + t_{eval}^{arrived} + t_{com}^{C_{ts},ts}) - t_{com}^{x,ts} \quad (1)$$

where t_{eval}^e is the time it takes a controller to compute and carry out the ruling for event `e`, and $t_{com}^{a,b}$ is the communica-

tion time from `a` to `b`. The *relative overhead* $ro_{x,ts}$ of an LGI message from `x` to `ts`—relative to the direct transmission of such a message—is defined as:

$$ro_{x,ts} = o_{x,ts} / t_{com}^{x,ts} \quad (2)$$

When evaluating these formulae in specific situations we will use the following approximations and typical values. First, the communication time $t_{com}^{a,b}$ depends on many factors, including the length of message, the communication protocol being used, the distance between the communicating parties, and whether the message is sent in clear or signed. We will ignore many of these factors, and distinguish only between the following quantities: (We specify, within parenthesis, the typical value we will be using for each of them.)

1. t_{pipe} (≈ 0.1 ms): the communication time via a pipe, for `a` and `b` that reside on the same machine.
2. t_{WAN} (≈ 50 ms): the TCP/IP communication time, for `a` and `b` residing in different LANs (the message is not signed).
3. t_{signed} (≈ 100 ms): the communication time for `a` and `b` communicating via signed messages across a WAN—it takes into account the time required to sign the message and to verify the signature.

Second, the experiments we performed showed that the evaluation time is relatively insensitive to the event; as such we will use the approximation $t_{eval}^e \approx t_{eval}$. The time taken by our current, experimental controllers to evaluate an event is 3.5ms (see Section 5.2).

The LGI model is silent on the placement of controllers *vis-a-vis* the agents they serve, and it allows for the sharing of a single controller by several agents. Moreover, the current implementation supports both clear and signed communication between parties⁶. This provides us with flexibilities, which can often be used to minimize the overhead of LGI under various conditions. We will consider here in detail the effect of these factors on the relative overhead of LGI across a wide area network (WAN).

5.1.1 Using Local Controllers

Perhaps the most natural way to use LGI, and usually the most efficient one, is to place each controller C_x at the host machine of agent `x` itself, as illustrated in part (a) of Figure 4. This allows each agent to communicate with its controller via pipes, which is substantially more efficient than TCP communication. Applying Equations 2 and 1 to this situation and assuming that the controller-controller communication is not signed, yield the following result for relative overhead:

$$ro_{x,ts} = (2 * t_{eval} + 2 * t_{pipe}) / t_{WAN} \approx 0.14 \quad (3)$$

This overhead is quite negligible. However, as argued below, this scheme can be used only when clients of the tuple space are assumed to be non-malicious.

⁶In the current implementation, the secretary decides whether the communication should be in clear or signed. Due to lack of space we are unable to present here the authentication protocols (the reader is referred to [15]). Sufices to say that they ensure that an agent cannot masquerade as another agent or as a controller.

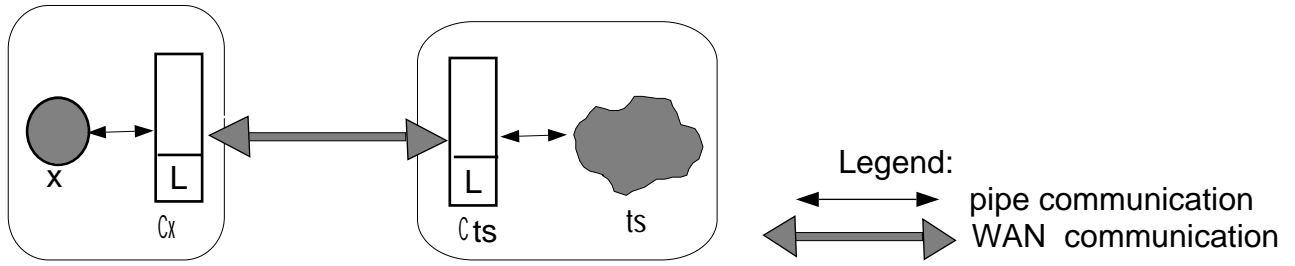


Figure 4: Controllers are placed on the same machine as the agents and the communication is done in clear.

5.1.2 Using Remote Controllers

The above scheme has the following problems when one is concerned that agents may be malicious. First, local controllers can be tempered with by their clients. Second, if clear communication is used, an agent can masquerade as another agent or as a controller. The security is enhanced if controllers are placed on trusted machines, and messages are signed when sent over the network. Such controllers would generally not reside in the LAN of their clients, but might be anywhere in the Internet.

To compute the relative overhead of such communication, illustrated in part (a) of Figure 5, we plug t_{signed} for every communication time in Equation 1. This yields the following result for the relative overhead in this case:

$$ro_{x,y} = (2 * t_{signed} + 2 * t_{eval}) / t_{signed} \approx 2 \quad (4)$$

The last step is justified by the fact that t_{eval} is numerically so much smaller than t_{signed} .

Although this overhead is not negligible, it is not prohibitive. Furthermore, as we shall see in the following section, even when security is an issue it is often possible to dramatically reduce this overhead by exploiting the ability of several agents to share a single controller.

5.1.3 Sharing Controllers

Suppose that a single controller C is assigned to the tuple space ts and to all agents x_1, \dots, x_n operating under a certain policy. Since the tuple space is trusted, we can place C on the same machine as the tuple space and consequently, the communication between them can be done via pipes. The processing with such a controller of a regulated message from x_i to ts , where i belongs to $1, \dots, n$ is illustrated in part (b) Figure 5. The controller-to-controller message disappears now, but we still have two evaluations of the law, one for the *sent*-event and one for *arrived*-event⁷. As such, this placement scheme requires only one more message than required by unregulated message passing. Our formula for relative overhead would now yield

$$ro_{x_i,ts} = (t_{pipe} + 2 * t_{eval}) / t_{signed} \approx 0.07 \quad (5)$$

This is a very low overhead for all communication with the tuple space. However, such a solution is not scalable especially if the number of participants in a policy is large, or the message-traffic is high.

⁷Controller-sharing works as follows: each controller maintains a table with all agents currently assigned to it. When a controller has to forward a message m to an agent y , it first looks for y in the table of assigned members. If the look-up is successful, the controller simply places the corresponding *arrived*-event in the y 's queue.

5.2 The Moses Toolkit and its Performance

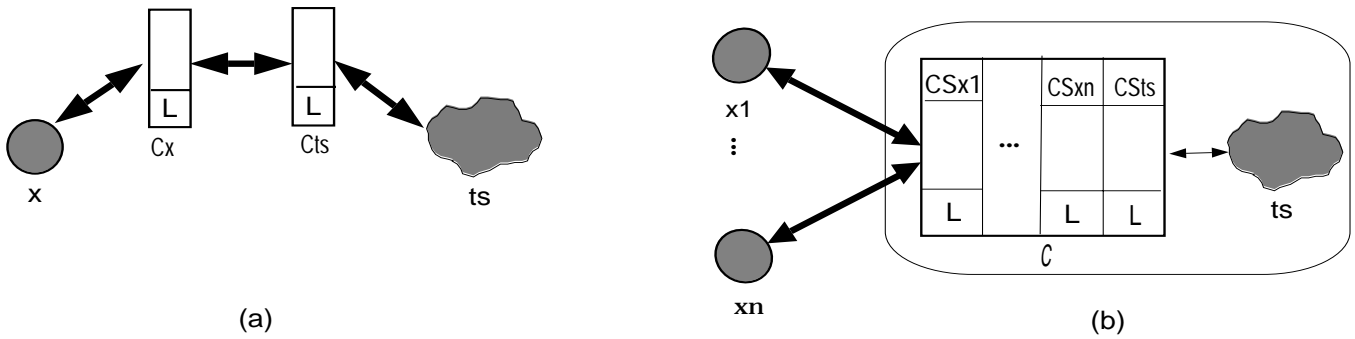
We have conducted many experiments with Moses, in a variety of configurations. Most of these experiments are difficult to summarize neatly because of the large variations in communication times. We report here in detail only our measurement of the performance of the controller itself, in isolation of other factors like network load or the response time of different agents. We were particularly interested in the *robustness* of the controller, when it has to deal with many different clients, and with many events. In order to do so, we measured the average *throughput* of a controller, i.e., the number of processed events per second, as a function of the number of clients handled by the controller. The experiment was conducted on a SUNW, Ultra-2 machine operating at 296 MHz, using Solaris 2.6 operating system and Java 1.2.

The experiment consists of several runs. In each run n_e *sent* events are processed, and the throughput is computed as n_e / t_p , where t_p is the *real time* it took the controller to process these events. Variant in the runs is n , the number of clients assigned to the controller. Figure 6 shows the results of the experiment with n ranging between 1 and 10, and $n_e=3000$. The experiment shows significant variations in the throughput—it ranges between 290 events/s and 390 events/s—with the number of clients. This gain in performance is due to the controller capacity to parallelize tasks: if all messages originate from a single member the evaluation of events is strictly sequential. If messages are sent by different agents, their evaluation proceeds in parallel. The best performance of the controller is reached for $n=5$; when the number of agents assigned to a controller is increased further, the bookkeeping time offsets some of the parallelism gain.

6. RELATED WORK

We already discussed, in the introduction, two kinds of approaches to make tuple spaces safe: traditional ACL-based access control, as in IBM Tspaces, and the use of multiple tuple spaces. We found both of these approaches wanting. Here we will consider three additional approaches.

The Klaim language [5] enhances the safety of tuple spaces via strong typing. Access to tuple is regulated here via *typed access rights*, making it possible to determine statically, and thus very efficiently, whether an access is allowed or not. This method can be used to prevent many inadvertent errors by buggy code. However, it does not provide effective means for protection against potentially *malicious* agents, since it is not possible to rely on typing if the requests come from untrusted sites.



Legend:

 pipe communication
 signed communication

Figure 5: (a) Controllers are placed remote (across a WAN); (b) Agents x_1, \dots, x_n and tuple space ts share the same controller C .

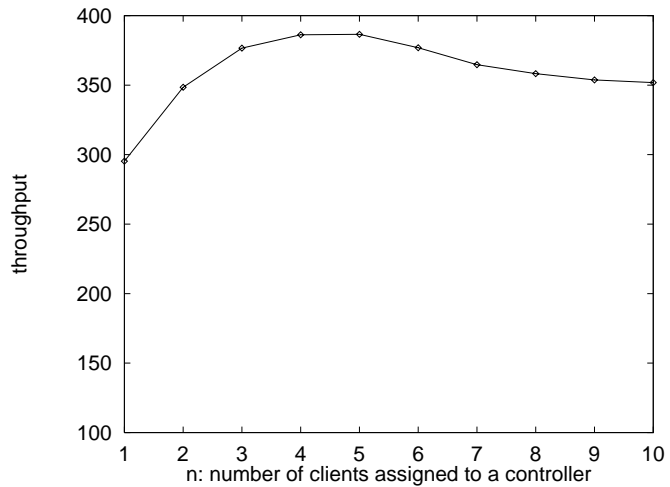


Figure 6: The performance of a controller under a heavy load

The SecOS [2] system attempts to provide secure access to tuple spaces via explicit cryptographic techniques. Under this system, entries in a tuple space are encrypted, and can be decrypted only by agents holding the correct key. This scheme has several serious limitations. First, it does not protect a tuple from being deleted from the tuple space, even by agents that do not have keys for some fields in it. Indeed, any tuple can be removed by anybody, simply by issuing an `in` command, with an empty template, for example. Second, this scheme leaves it up to the originator of every tuple to distribute keys for it. Such key distribution may become a managerial nightmare, especially when dealing with large and rapidly changing group of agents.

Perhaps the closest work to this paper is *programmable tuple space* [6; 7], called LuCe. Like us, they call for an explicit formulation of a policy, which is to be written in a formal language. The programmability is achieved by triggering a reaction whenever a communication event occurs. A reaction, similar to our rule, consists of a set of primitive

operations to be executed when the event occurs. A major difference between Moses and LuCe is that the latter does not maintain state for the clients of the tuple space. Such state is necessary for many security policies, such as the congestion control policy discussed in Section 4.2, or the well known Chinese Wall policy [1].

7. CONCLUSION

Our objective in this paper has been to remedy the lack of safety inherent in using tuple space based middleware for open systems. We have demonstrated how a law-governed interaction can be used to add a wide variety of guarantees to a tuplespace without eliminating the flexibility that makes tuplespaces attractive in the first place. Moreover, these guarantees can be added transparently, allowing them to be integrated into an existing system.

LGI-based control is particularly appropriate for use in the context of tuplespaces for the following reasons:

- **Laws under LGI are sensitive to the content of the tuples being handled.** This property is an essential part of what makes it possible to implement useful guarantees efficiently under our regime. It is in particular required for both examples in this paper.
- **Laws are sensitive to the state of agents,** which can change dynamically. This property is critical to the congestion control policy, and to many others.
- **Enforcement of laws can occur at either the client, the server, or anywhere in the network.** Pushing enforcement responsibilities to the client can greatly improve scalability. Efficiency aside, some kinds of restrictions, such as congestion control, cannot be enforced at the server.

8. ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their helpful comments on the previous version of this paper. This work was supported in part by NSF grants CCR-96-26577, CCR-97-10575 and CCR-98-03698.

9. REFERENCES

- [1] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium in Security and Privacy*. IEEE Computer Society, 1989.
- [2] C. Bryce, M. Oriol, and J. Vitek. A coordination model for agents based on secure spaces. In P. Ciancarini and A. L. Wolf, editors, *Proc. of Coordination'99: Third International Conference on Coordination Models and Languages; LNCS 1594*, pages 4–20, April 1999.
- [3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.
- [4] P. Ciancarini. Enacting rule-based software processes with polis. Technical report, University of Pisa, october 1991.
- [5] R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In D. Garlan and D. L. Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 221–237, September 1997.
- [6] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. L. Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 274–288, September 1997.
- [7] E. Denti and A. Omicini. An architecture for tuple-based coordination of multi-agent systems. *Software—Practice & Experience*, 29(12):1103–1121, 1999.
- [8] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces(TM) Principles, Patterns and Practice (The Jini(TM) Technology Series)*. Addison-Wesley, 1999.
- [9] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
- [10] Jini. Technical report, Sun Microsystems. website: <http://java.sun.com/products/jini/>.
- [11] N. Minsky and J. Leichter. Law-governed Linda as a coordination model. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, number 924 in Lecture Notes in Computer Science, pages 125–146. Springer-Verlag, 1995.
- [12] N. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, May 1998.
- [13] J. Pinakis. Providing directed communication in Linda. In *Proceedings of the 15th Australian Computer Science Conf.*, pages 731–743, 1992.
- [14] P. Tarau. Language issues and programming techniques in BinProlog. In *Proceedings of of the Gulp'93 Conference*, June 1993.
- [15] V. Ungureanu. *A Mechanism for Supporting Communication Policies in Distributed Systems*. PhD thesis, Rutgers University, 2000. Obtainable from ungurean@cs.rutgers.edu.
- [16] P. Wyckoff, W. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Journal of Research and Development*, 37:454–474, 1988.