Safe Tuplespace-Based Coordination in Multi Agent Systems

Naftaly H. Minsky* Yaron M. Minsky† Victoria Ungureanu⋆

Abstract

Linda is a high level communication model which allows agents to communicate via a shared tuplespaces 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 tuplespaces, this paper introduces a mechanism for establishing security policies that regulate agent access to tuplespaces. 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 our 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 tuplespaces.

1 Introduction

In the mid-80's, the Linda Project [4] 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 tuplespace) to the task of building middleware for heterogeneous distributed systems, or *open* systems.

Interest in the use of tuplespace like abstractions for open systems has gained attention recently with the introduction of systems like Sun's *JavaSpaces* [10] and IBM's *TSpaces* [27], which marry a Linda-like tuplespace to the Java programming 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 [12].

There is much to be said for the use of Linda as a coordination model for open systems [11]. 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 tuplespaces makes it unsafe for use in open systems. This is clearly a security concern: a malicious client with access to a given tuplespace could disrupt any system that depends on the integrity of the data stored in that tuplespace. But even when security is not an issue i.e., when all clients of a given tuplespace are assumed to be non-malicious, the use of unprotected tuplespaces is still a threat to system stability. A buggy agent

*`minsky@cs.rutgers.edu`, Department of Computer Science, Rutgers University, New Brunswick, NJ, 08903 USA.
†`yminsky@cs.cornell.edu`, Department of Computer Science, Cornell University, Ithaca, NY.

could easily corrupt a shared tuplespace, thus disrupting the activities of the other clients of the tuplespace. 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 [5], among many others, enhanced Linda by making it support a sophisticated multiple tuplespace organization. Both JavaSpaces and TSpaces have adopted the use of multiple tuplespaces, and provide simple access control on a per tuplespace basis.

However, segregating communication into multiple tuplespaces increases safety only insofar as it eliminates sharing. But tuplespaces are most useful when diverse agents share access to a single tuplespace. Moreover, the content-based nature of retrieval from tuplespaces 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 tuplespace.

To provide for such content-based access control, a new model for Linda, called law-governed Linda, has been proposed in [16], but had never been fully implemented. In this paper we show that a similar, but more powerful, control mechanism can be established without changing the Linda model itself[1], by subjecting the interaction of tuplespaces 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 tuplespaces, 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 tuplespaces. In Section 3 we provide a summary of LGI, and explain briefly how it is applied to tuplespaces. 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 [25]. 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 tuplespace 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 tuplespace. 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.

[1]The model described in [16] involved a substantial change in the Linda mechanism

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

1. Any agent interacting with the tuplespace 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-tuplespaces, 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 [19], 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)—an Overview

Broadly speaking, LGI is a mode of interaction that allows an heterogeneous group of distributed agents to interact with each other, *with confidence that an explicitly specified set $\mathcal{L}$ of rules of engagement—called the law of the group—is complied with.* A group of agents thus interacting via LGI under a given law $\mathcal{L}$, is called an $\mathcal{L}$-group. This mode of interaction is currently supported by a toolkit called Moses, which is implemented mostly in Java.

We provide in this section a brief overview of LGI; for more detailed discussion see [18]. The description is organized as follows: We start in Section 3.1 by formally defining the concept of an $\mathcal{L}$-group. In Section 3.2 we present the other basic elements of LGI. The law-enforcement mechanism is discussed in Section 3.3. Our language for specifying laws is presented in Section 3.4, and its use is illustrated by formalizing the message passing policy presented in the previous Section. In Section 3.5 we present an additional feature of LGI—obligations; we conclude with a discussion of the levels of security provided by the current implementation of LGI.

3.1 The Concept of an $\mathcal{L}$-Group

An $\mathcal{L}$-group $\mathcal{G}$ can be defined as the four-tuple $\langle \mathcal{L}, \mathcal{A}, \mathcal{CS}, \mathcal{M} \rangle$ where,

1. $\mathcal{L}$—the *law* of the group—is an *explicit and enforced* set of "rules of engagement" between members of this group.

2. $\mathcal{A}$ is the set of *agents* belonging to $\mathcal{G}$—the *members* of this group.

3. $\mathcal{CS}$ is a set $\{\mathcal{CS}_x \mid x \textbf{ in } \mathcal{A}\}$ of *control states*, one per member of the group. $\mathcal{CS}$ is mutable, subject to law $\mathcal{L}$ of the group.

4. $\mathcal{M}$ is the set of messages that can be exchanged, under law $\mathcal{L}$, between members of $\mathcal{G}$—they are called $\mathcal{L}$-messages.

We will now elaborate on the components of an $\mathcal{L}$-group.

The Law: The law is defined over certain types of events occuring at members of $\mathcal{G}$, mandating the effect that any such event should have—this mandate is called the *ruling* of the law for a given event. The events thus subject to the law of a group under LGI are called *regulated events*—they include (but are not limited to) the sending and arrival of $\mathcal{L}$-messages.

The law of a given group $\mathcal{G}$ is *global* with respect to $\mathcal{G}$, but it is defined *locally* at each member of it. The law is global, in that *all* members of the group are subject to it; and it is *defined locally*, at each member, in the following respects:

- The law regulates explicitly only *local events* at individual agents.
- The ruling of the law for an event `e` at agent `x` depends only on `e` itself and on the *local control-state* $\mathcal{CS}_x$ of `x`.
- The ruling of the law at a given agent `x` can mandate only *local operations* to be carried out at `x`, such as an update of the local *control-state* $\mathcal{CS}_x$, or the forwarding of a message from `x` to some other agent.

Note that it is the globality of law $\mathcal{L}_\mathcal{G}$ that establishes a *common* set of ground rules for all members of $\mathcal{G}$, providing them with the ability to trust each other, in spite of the heterogeneity of the group. And it is the locality of the law that enables its scalable enforcement, by means of a trusted agent called *controller* associated with individual members of the group.

Abstractly speaking, the law $\mathcal{L}$ of a group 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 in response to the event in question, at its home. (An empty ruling simply implies that the event in question has no consequences—such an event is effectively ignored.) Later we will introduce the language we use for specifying such laws in our current implementation of LGI. But the nature of this language is, in a sense, of a secondary importance.

The Group: We refer to members of an $\mathcal{L}$-group as *agents*, by which we mean autonomous actors that can interact with each other, and with their environment. Such an agent might be an encapsulated software entity, with its own state and thread of control, or it might be a human that interacts with the system via some interface. (Given popular usage of the term "agent", it is important to point out that this term does not imply here either "intelligence" nor mobility, although neither of these is ruled out.) Nothing is assumed here about the structure and behavior of the members of a given $\mathcal{L}$-group, which are viewed simply as sources of messages, and targets for them.

The Control-States: For each agent `x` in $\mathcal{G}$, LGI maintains the *control-state* $\mathcal{CS}_x$ of this agent, whose semantics, for a given $\mathcal{L}$-group, is defined by its law. Typically, the control-state of an agent could represent such things as the role of this agent, special privileges it has under this law, and various kinds of tokens it carries—and it can change dynamically, subject to the law.

Control-state $\mathcal{CS}_x$ is not directly accessible to agent `x` (or to any other agent). It is maintained by the controller assigned to `x`, and can be changed only by operations included in the ruling of the law for events at `x`. Structurally, $\mathcal{CS}_x$ is a bag of Prolog-like terms, called the *attributes* of agent `x`, whose meaning is defined by the law of any given group.

3.2 Additional Elements of LGI

Regulated Events: The events that are subject to laws are called *regulated events*. Each such event is viewed as occuring at a certain agent `h`, called the *home* of the event—strictly speaking, however, events occur at the controller $\mathcal{C}_h$ assigned to their home. We introduce here three types of regulated events. The first pair of events represents stages of the passing of an $\mathcal{L}$-message. The last event, which will be discussed in detail only in Section 3.5, deals with *obligations*.

Operations on the control-state	
`t@CS`	returns true if term `t` is present in the control state, and fails otherwise
`+t`	adds term `t` to the control state;
`-t`	removes term `t` from the control state;
`t1←t2`	replaces term `t1` with term `t2`;
`incr(t(v),d)`	increments the value of the parameter `v` of term `t` with quantity `d`
`dcr(t(v),d)`	decrements the value of the parameter `v` of term `t` with quantity `d`

Operations on messages	
forward(x,m,y)	sends message `m` from `x` to `y`; triggers at `y` an `arrived(x,m,y)` event
deliver(x,m,y)	delivers to agent `y` message `m` (sent by `x`)

Figure 1: Some Primitive Operations

1. `sent(h,m,y)`—occurs when an $\mathcal{L}$-message `m` sent by `h` to `y` arrives at $\mathcal{C}_h$. (The sender `h` is the *home* of this event.) The destination `y` of the message `m` can be either the name of a specific member in $\mathcal{G}$, or a list of such names, which allows for multicasting.

2. `arrived(x,m,h)`—occurs when an $\mathcal{L}$-message `m` ostensibly[2] sent by `x`, arrives at $\mathcal{C}_h$. The receiver `h` is the *home* of this event.

3. `obligationDue(...)`—the occurrence of this event means that it is time to enforce an *obligation* previously imposed on the home of this event. (Obligations are discussed in Section 3.5).

It should be pointed out that this is not a complete set of regulated events. Our current LGI mechanism features several additional types of regulated events—dealing with interoperability between laws, the import of certificates, and other matters—which are beyond the scope of this paper.

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.* Primitive operations currently supported by LGI include operations for testing the control-state of an agent and for its update, operations on messages, and some others. A sample of primitive operations is presented in Figure 1.

3.3 The Distributed Law-Enforcement Mechanism

The law $\mathcal{L}$ of an $\mathcal{L}$-group $\mathcal{G}$ is enforced by a set of trusted agents called *controllers*, that mediate the exchange of $\mathcal{L}$-messages between members of the group. Every member `x` of $\mathcal{G}$ has a controller $\mathcal{C}_x$ assigned to it, which maintains the control-state $\mathcal{CS}_x$ of its client `x`. And all these controllers, which are logically placed between the members of group $\mathcal{G}$ and the communications medium, carry the *same law* $\mathcal{L}$ (as illustrated in Figure 2). This allows the controller $\mathcal{C}_x$ assigned to `x` to compute the ruling of $\mathcal{L}$ for every event at `x`, and to carry out this ruling locally.

Controllers are *generic*, and can interpret and enforce any well formed law. A controller operates as an independent process, and it may be placed on the same machine as its client, or on some other machine, anywhere in the network. Under Moses (our current implementation of LGI) each controller can serve several agents, operating under possibly different laws. This facilitates various optimization techniques, discussed in Section 5.

In the current implementation an $\mathcal{L}$-group, $\mathcal{G}$ is maintained by a server that provides persistent storage for the law $\mathcal{L}$ of this group and the control-states of its members. This server is called the *secretary* of $\mathcal{G}$, to be denoted by $\mathcal{S}_{\mathcal{G}}$. For an agent `x` to be able to exchange $\mathcal{L}$-messages under a

[2] The actual sender of this message may be other than `x`, as the law under LGI has the power to *misrepresent* the sender—which is useful in some cases.

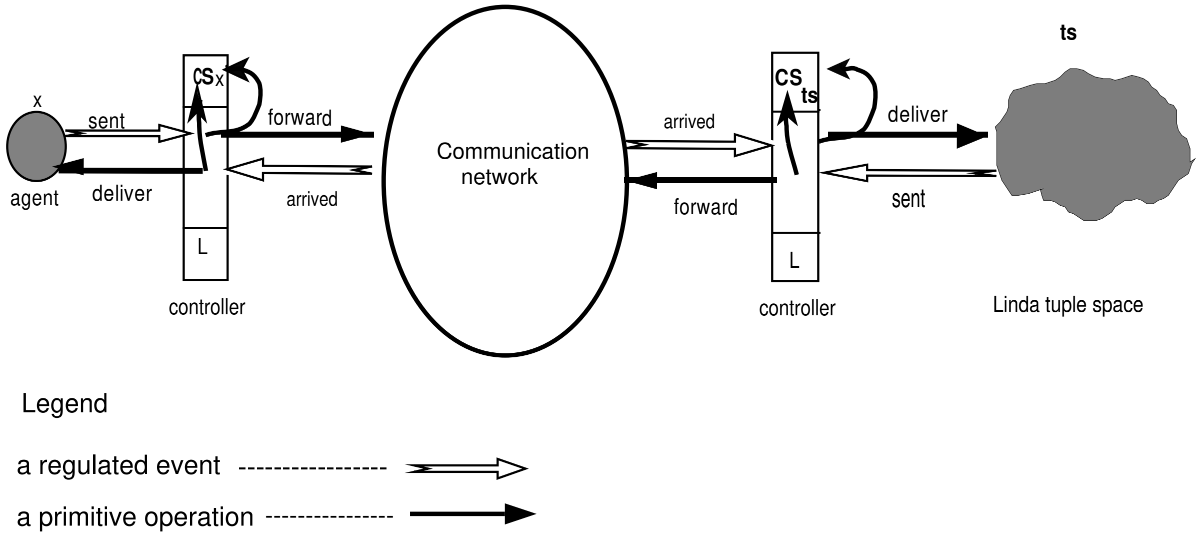

Figure 2: Enforcement of the Law

law $\mathcal{L}$, it needs to engage in a connection protocol with the secretary. The purpose of the protocol is to assign `x` to a controller $\mathcal{C}_x$ which is fed the law of $\mathcal{G}$ and the control state of `x` (for a detailed presentation of this protocol the reader is referred to [18]).

We are in position now to explain how the exchange of $\mathcal{L}$-messages gets to be mediated by controllers, and how this mediation is carried out. Consider, for example, an agent `x` sending a $\mathcal{L}$-message `m` to a tuplespace `ts`, assuming that both `x` and `ts` have joined group $\mathcal{G}$. Message `m` is sent by means of a routine provided by the Moses toolkit, which forwards it to $\mathcal{C}_x$—the controller assigned to `x`. When this message arrives at $\mathcal{C}_x$, it generates a `sent(x,m,ts)` event at it. $\mathcal{C}_x$ then evaluates the ruling of law $\mathcal{L}$ for this event, taking into account the control-state $\mathcal{CS}_x$ that it maintains, and carries out this ruling.

If this ruling calls the control-state $\mathcal{CS}_x$ to be updated, such update is carried out directly by $\mathcal{C}_x$. And if the ruling calls for message `m` to be forwarded to `ts`, then $\mathcal{C}_x$ would send `m` to the controller $\mathcal{C}_{ts}$ assigned to `ts`. This is done as follows: if $\mathcal{C}_x$ does not have the address of $\mathcal{C}_{ts}$ it will ask the secretary. When the secretary responds, $\mathcal{C}_x$ will finalize the `forward` and will cache the address. As such, forthcoming communication between `x` and `ts` will not require the extra step of contacting $\mathcal{S}_{\mathcal{G}}$.

When the message `m` sent by $\mathcal{C}_x$ arrives at $\mathcal{C}_{ts}$ it generates an `arrived(x,m,ts)` event. Controller $\mathcal{C}_{ts}$ 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 $\mathcal{CS}_{ts}$ maintained by $\mathcal{C}_{ts}$ to be modified.

In general, all regulated events that occur nominally at an agent `x` actually occur at its controller $\mathcal{C}_x$. 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 occuring at `x`. Note that a controller might be associated with several agents, in which case events pertaining to different agents are evaluated concurrently.

3.4 The Formulation of Laws

Laws can be quite naturally expressed by mean of any language based on *event-condition-action* (ECA) kind of rules. For now, we have chosen a somewhat restricted version of Prolog [6], due to its expressive power, and its relatively widespread usage. Under the Moses implementation of LGI, then, the law is defined by means of a Prolog-like program `L` which, when presented with a goal `e`, representing a regulated-event at a given agent `x`, evaluates it in the context of the control-state of this agent. This evaluation produces a list of primitive-operations representing the ruling of the law for this event. In addition to the standard types of Prolog goals, the body of a rule may contain a distinguished type of goal that contribute to the ruling of the law. It has the form `do(p)`, where `p` is one of the above mentioned primitive-operations and it appends term `p` to the ruling of the law.

Message Passing Policy We will not be able to present here additional aspects of law formulation—the reader is referred to [18] or to [17] for more details—but we will complement this discussion with the presentation of law $\mathcal{L}_{\mathcal{MP}}$ that implements the message passing policy introduced informally in Section 2.

Formally, under LGI, the components of $\mathcal{MP}$-group are as follows: the group $\mathcal{G}$ consists of the tuplespace servers[3] and their clients. The set $\mathcal{M}$ of messages regulated by $\mathcal{L}_{\mathcal{MP}}$ consists of: `out([msg(m),from(s),to(t)]` and `in[msg(m),from(s),to(t)]` denoting requests to post to, and respectively retrieve from a tuplespace a message `m`. The control-state of each tuplespace server contains a term `tupleSpace` which denotes its role. (Clients have empty control states.) Finally, the law of this policy is presented in Figure 3. This law consists of four rules. Each rule is followed by a comment (in italic), which, together with the following discussion, should be understandable even for a reader not well versed in Prolog.

Initially: tuplespace servers have in their control state a term `tupleSpace`.

$\mathcal{R}1$. `sent(X,out([msg(M),from(X'),to(Y)],TS) :- X==X',do(forward).`

Any agent can `out` *a* `msg` *tuple, if identified as being from itself.*

$\mathcal{R}2$.
```
sent(Y,in([msg(M),from(X),to(Y')]), TS) :-
        Y==Y', do(forward).
```

An agent `Y` *may* `in` *only messages meant for itself.*

$\mathcal{R}3$. `sent(TS,_,_) :- tuplsSpace@CS, do(forward).`

Any message sent by a tuplespace is forwarded.

$\mathcal{R}4$. `arrived(_,_,_) :- do(deliver).`

When a message arrives anywhere, it is delivered.

Figure 3: Law $\mathcal{L}_{\mathcal{MP}}$—message passing policy

Message posting to tuplespaces is regulated by Rule $\mathcal{R}1$, which mandates that a message is to be forwarded to the intended tuplespace only if the value of `from` field of is identical with the id of the sender of the message. Thus, this rules ensures that sender's name cannot be forged.

Rule $\mathcal{R}2$ provide for safe retrieval of messages from the tuplespace. By this rule, an `in` request is forwarded to the intended tuplespace only if the id of the retriever coincides with the value of the `to` field. Thus, this rule ensures that a message can be fetched only by the agent for which it was intended.

Rules $\mathcal{R}3$ and $\mathcal{R}3$ regulate the treatment of messages arriving at a tuple server and its consequent responses. Since the access control policy is exercised at the client side, these messages are delivered, and respectively forwarded without further ado. Finally, when a response arrives at a client it is delivered (Rule $\mathcal{R}3$).

3.5 A concept of enforceable obligation

Obligations are widely considered essential for the specification of enterprise policies, along with permissions and prohibitions. The concept of obligation being employed for this purpose is usually based on conventional *deontic logic* [15], designed for the specification of normative systems, or on some elaborations of this logic, such as taking into account interacting agents [2]. These types of obligations allows one to reason about what an agent must do, but they provide no means for

[3] note, that the tuplespace itself may be is distributed

ensuring that what needs to be done will actually be done [14, 13, 23, 21]. LGI, on the other hand, features a concept of obligation that can be enforced.

Informally speaking, an obligation under LGI is a kind of *motive force*. Once an obligation is imposed on an agent—which can be done as part of the ruling of the law for some event—it ensures that a certain action (called *sanction*) is carried out at this agent, at a specified time in the future, when the obligation is said to *come due*—provided that certain conditions on the control state of the agent are satisfied at that time. The circumstances under which an agent may incur an obligation, the treatment of pending obligations, and the nature of the sanctions, are all governed by the law of the group. An example of the use of obligations under LGI is given in Section 4.2.

Specifically, suppose that at time t_0, an agent `x` incurs an obligation by the execution at `x` of a primitive operation `imposeObligation(oType,dt)` where `dt` is the time period, expressed in seconds, after which the obligation is to come due; and `oType`—the *obligation type*—is a term that identifies this obligation (not necessarily in a unique way). The main effect of this operation is that unless the specified obligation is *repealed* (see below) before time $t = t_0 + dt$, the *regulated event* `obligationDue(oType)` would occur at agent `x` at time `t`. The occurrence of this event would cause the controller to carry out the ruling of the law for this event; this ruling is thus the *sanction* for this obligation. (Note that all the times here are defined by the local clock of agent `x`.)

A pending obligation incurred by agent `x` can be *repealed* before its due time by means of the primitive operation `repealObligation(oType)` carried out at `x`, as part of a ruling of some event. This operation actually repeals *all* pending obligations of type `oType`. We have more to say about this concept, after we illustrate its use with an example.

3.6 Security Dimensions

Broadly speaking, one can distinguish between two types of potential violations of a given law: (a) *inadvertent* violations, due to a bug in the code of an agent, say, or due to its ignorance of the law; and (b) *malicious* violations. It is our assumption that less stringent security requirements will apply to the former scenario than to the latter. Consequently, Moses provides for an array of protection tools, and for flexibility in terms of choosing them.

To ensure that inadvertent violations do not occur, it is sufficient to have the controller be a small and carefully coded piece of code that is used standardly by all clients. And to ensure that the various controllers agree on the nature of a given law $\mathcal{L}$, $\mathcal{L}$-messages sent across the network include a hash of the law. (The hash of the law is obtained using one way function which transforms any string into a considerably smaller bits sequence with high probability that two strings will not collide. Specifically, we are using MD5 algorithm [22] for computing the hash.)

To prevent malicious violations of the law, the following conditions have to be met: (1) messages are securely transmitted over the network, and (2) $\mathcal{L}$-messages are sent and received only via correctly implemented controllers, interpreting law $\mathcal{L}$. To ensure the first of these conditions, every agent belonging to an $\mathcal{L}$-group $\mathcal{G}$, and each controller, have a pair of keys: a public key known to a trusted authority and a secret key known only by itself. If the messages sent across the network are digitally signed their authenticity is guaranteed as long as the private key is not disclosed. (In the current implementation we are using RSA algorithm [20] for computing digital signatures.)

Condition (2) above is more problematic, and can be handled at different levels of security. First, controllers may be placed on trusted machines, like for example the servers on which the tuplespaces reside. Then, a controller can only be compromised if the tuplespace itself is compromised. Second, controllers may be trusted when built into *physically secure coprocessors* [28, 24]. Such a secure device consists of a CPU, non-volatile memory, encryption hardware and special sensing circuitry to detect intrusion. The sensing circuitry erases non-volatile memory before attackers can penetrate far enough to disable the sensors or read memory contents. If controllers are implemented on such physically secure hardware devices, the receiver of an $\mathcal{L}$-regulated message has a high degree of trust that this message is authentic, in the sense that the message has been sent by a genuine controller interpreting law $\mathcal{L}$.

4 Examples

We present here two quite different examples of LGI-based coordination via tuplespaces. The first example is that of a policy that supports secure interaction between service providers and their customers. Our second example shows how a tuplespace 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 tuplespace `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 *out*s into `ts` a *request-tuple*

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

identifying itself and the service requested; (b) service provider bid for this service by *out*ing *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 tuplespace (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 $\mathcal{L}_{\mathcal{SB}}$ in Figure 4. Let us examine now this particular law in detail:

- 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 rules 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 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).

Initially: Agents have in their control states terms denoting the role they play: `serviceProvider` for providers and `tupleSpace` for tuplespace servers.

$\mathcal{R}1$.
```
sent(C,out([requester(C'),service(S)]),TS) :-
        C==C',do(forward).
```
Any client `C` *can* `out` *request-tuples for services, if identified as being from himself.*

$\mathcal{R}2$.
```
sent(C, in([requester(C'),service(S)]),TS) :-
        C==C',do(forward).
```
Any client `C` *can* `in` *(i.e. read and remove) request-tuples previously posted by himself.*

$\mathcal{R}3$.
```
sent(P, rd([requester(C),service(S)]),TS) :-
        serviceProvider@CS, do(forward).
```
Any service provider `P` *can read requests posted to a tuplespace.*

$\mathcal{R}4$.
```
sent(P1, out([offerFor(C,S), fee(F),provider(P2),contact(Addr)]),TS) :-
        P1==P2, serviceProvider@CS, do(forward).
```
Any service provider `P` *can* `out` *offer-tuples, if identified (in term* `provider`*, as being from himself.*

$\mathcal{R}5$.
```
sent(C,in([offerFor(C',S), fee(F), provider(P),contact(Addr)]), TS) :-
        C==C', do(forward).
```
Only the client who posted a service request can `in` *a corresponding offer.*

$\mathcal{R}6$.
```
sent(TS,_,_) :- tupleSpace@CS, do(forward).
```
Any message sent by a tuplespace is forwarded.

$\mathcal{R}7$.
```
arrived(_,_,_) :- do(deliver).
```
when a message arrives anywhere, it is delivered.

Figure 4: Law $\mathcal{L}_{\mathcal{SB}}$ of Secure Bidding

- Finally, Rule $\mathcal{R}6$ allows the tuplespace `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 tuplespace to protect itself from getting congested, by controlling the frequency of messages sent to it by its users:

1. Every client of a tuplespace `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 tuplespace.
2. The server of the tuplespace can set the delay of an agent to any desired value.
3. If an agent attempts to send a message to `ts` sooner than permitted by his delay, this message is to be forwarded to the server at the earliest time consistent with the delay, *without client's involvement.*

This policy, established by law $\mathcal{L}_{\mathcal{CC}}$, presented in Figure 5, is implemented as follows: early messages are buffered in the control state of a client, and obligations are imposed to forward the messages in question at the earliest possible time consistent with the delay condition. While a client may send its requests to `ts` at any rate, these messages will be actually forwarded to the destination at the pace imposed by `ts`, and in their original order.

There are three possibilities to take into account when a client attempts to send a message. First if the delay condition is satisfied and there are no buffered messages, then the message is forwarded to the tuplespace `ts` (Rule $\mathcal{R}1$). Second, if the delay condition is not satisfied and the buffer is empty, then the message is pushed into the buffer and an obligation is imposed to forward the message at the earliest time that satisfies the delay condition (Rule $\mathcal{R}1$). Finally, if the delay condition is not satisfied and the buffer is not empty, the message is added to the buffer in FIFO order (Rule $\mathcal{R}2$). These rules, then, give the term `delay(dt)` in the CS of an agent the effect intended for it in our policy.

Now, by Rule $\mathcal{R}3$, when an obligation `sendMessage` fires the oldest message is removed from the buffer and is forwarded to `ts`. Moreover, if there are still buffered messages, then another `sendMessage` obligation is set for the earliest time satisfying the delay condition.

We end the presentation of this law by showing how the tuple space may adjust at will the `delay` term of his clients. By Rule $\mathcal{R}4$, 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 $\mathcal{R}6$[4]. But when a message `changeDelay(val)` sent by `ts` to a client `c`, then, by Rule $\mathcal{R}5$, the value of the term `delay` in the control state of `c` is set to `val`. Note that the `changeDelay` message affects 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.

Discussion Two observations about this policy are in order. First, note that for this policy to be effective, it must be enforced at the client side (by the client's controller, in our 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.

Second, the congestion policy as presented here, is devised for a system containing *only one* tuplespace. This is not, however, an intrinsic requirement: the policy can be easily extended for the case the tuplespace itself is distributed[5].

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.

The Relative Overhead of LGI Consider a message `m` sent by an agent `x` to a tuplespace `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 $\mathcal{C}_x$, (2) from $\mathcal{C}_x$ to $\mathcal{C}_{ts}$, and (3) from $\mathcal{C}_{ts}$ to `ts`. The overhead $o_{x,y}$, due to the extra messages and the law-evaluations involved, is given by the following formula:

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

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 communication 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:

$$ro_{x,ts} = o_{x,ts}/t_{com}^{x,ts} \tag{2}$$

[4]with 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))`.

[5]One possible implementation is to maintain different `delay`, `lastCall` and `buffer` for each tuplespace in the system

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 tuplespace `ts`; (2) a term `lastCall(Tlast)` where `Tlast` is the time when the last message was sent to the tuplespace (initially set to 0); and (3) a term `buffer(L)`, where `L` is the list of messages the client sent earlier than required by the delay condition and have not yet been forwarded (`L` is initially empty).

$\mathcal{R}1$.
```
sent(X,M,ts) :-
        buffer([])@CS, lastCall(Tlast)@CS,delay(DT)@CS,clock(T),
        T > (Tlast + DT) →
            (do(lastCall(Tlast)←lastCall(T)), do(forward))
            |
            (do(+obligation(sendMessage,Tlast+DT)),
            do(buffer([])←buffer([M]))).
```

A message sent to the server `s` *will be forwarded if there are not buffered messages and the delay condition is satisfied. Otherwise, the message is buffered and an obligation to send the message at the earliest time which satisfies the delay condition is set. In the case the message is forwarded the term* `lastCall` *is updated to reflect that a message was sent at the current time* `T`.

$\mathcal{R}2$.
```
sent(X,M,ts) :-
        buffer(L)@CS, append(L,[M],L1), do(buffer(L)←buffer(L1)).
```

If there are buffered messages, when a new message `M` *is sent to* `s`, `M` *is appended in FIFO order to the buffer.*

$\mathcal{R}3$.
```
obligationDue(sendMessage) :-
        lastCall(Tlast)@CS,delay(DT)@CS,clock(T)@CS,
        do(lastCall(Tlast)←lastCall(T)),
        buffer([M|R])@CS, do(buffer([M|R])←buffer(R)),
        do(forward(Self,M,ts)),
        R=[]→
            true
            |
            do(+obligation(sendMessage,T+DT)).
```

When an obligation `sendMessage` *fires the least recent message,* `M`, *is removed from the buffer and is forwarded to* `s`. *The term* `lastCall` *is updated to reflect that a message was sent at current time* `T`. *Moreover, if there are buffered messages an obligation* `sendMessage` *is set for the earliest time satisfying the delay condition.*

$\mathcal{R}4$.
```
sent(ts,_,_) :- do(forward).
```

Any message sent by the tuplespace `s` *is forwarded to its intended destination.*

$\mathcal{R}5$.
```
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`.

$\mathcal{R}6$.
```
arrived(_,M,_) :- not (M=changeDelay(V)),do(deliver).
```

Any message other than `changeDelay` *arriving at the destination is delivered without further ado.*

Figure 5: Law $\mathcal{L}_{CC}$ - Congestion Control Policy

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 the 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 residing on the same machine.
2. t_{WAN} (≈ 50ms): the TCP/IP communication time, for a and b residing in different LANs and the message is sent in clear.
3. t_{signed} (≈ 100ms): 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 its relatively insensitive on 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 (for a detailed presentation of the experiments see [18]).

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 parties. (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 [26]. Suffices to say that they ensure that an agent cannot masquerade as another agent or as a controller.) 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).

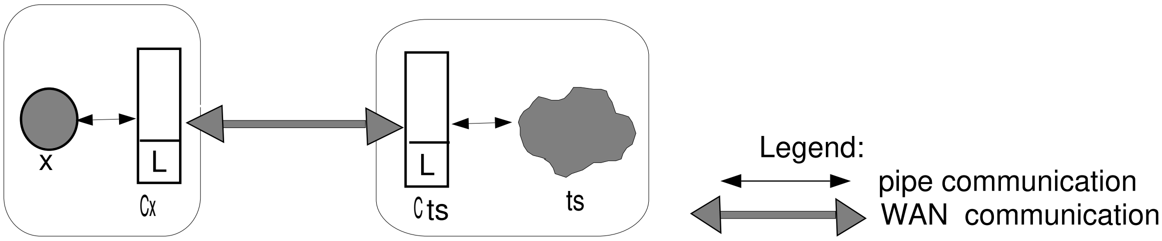

Figure 6: Controllers are placed on the same machine as the agents and the communication is done in clear.

Using Local Controllers Perhaps the most natural way to use LGI, and usually the most efficient one, is to place each controller $\mathcal{C}_x$ at the host machine of agent `x` itself, as illustrated in part (a) of Figure 6. 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:

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

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

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 7, we plug t_{signed} for every communication time in Equation 1. This yields the following result for the relative overhead in this case:

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

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.

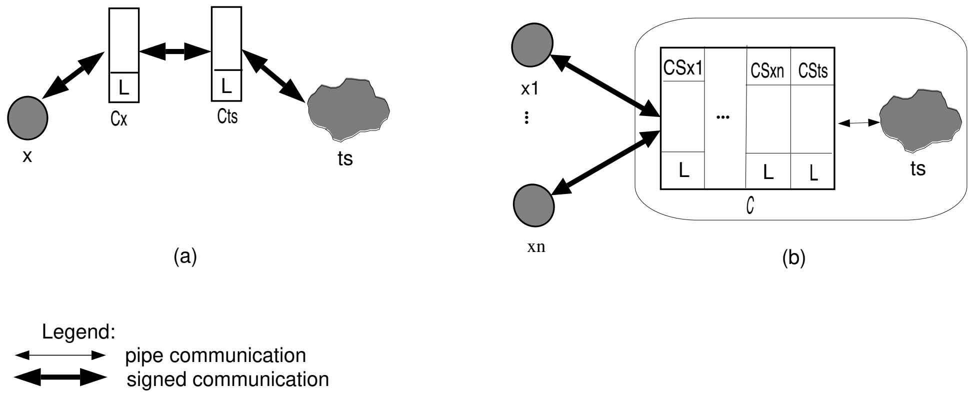

Figure 7: (a) Controllers are placed remote (across a WAN); (b) Agents $x1$,..., xn and tuplespace `ts` share the same controller *C*.

Sharing Controllers Suppose that a single controller C is assigned to the tuplespace `ts` and to all agents $x1, ..., xn$ operating under a certain policy. Since the tuplespace is trusted, we can place C on the same machine as the tuplespace and consequently, the communication between them can be done via pipes. The processing with such a controller of a regulated message from xi to `ts`, where i belongs to $1, .., n$ is illustrated in part (b) Figure 7.

Controller-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. As such,in this scheme, the controller-to-controller message disappears, but we still have two evaluations of the law, one for the *sent*-event and one for *arrived*-event. This placement technique requires only one more message than required by unregulated message passing. Our formula for relative overhead would now yield:

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

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.

6 Related Work

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

The Klaim language [7] enhances the safety of tuplespaces 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.

The SecOS [3] system attempts to provide secure access to tuplespaces via explicit cryptographic techniques. Under this system, entries in a tuplespace 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 tuplespace, 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 tuplespace* [8, 9], 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 tuplespace. 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 tuplespace 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.

- **LGI supports different levels of security.** Since the type of communication is not built-in, the mechanism can support in and efficient manner, policies with different security requirements.

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.

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] M. Brown. Agents with changing and conflicting commitments: a preliminary study. In *Proc. of Fourth International Conference on Deontic Logic in Computer Science (DEON'98)*, January 1998.

[3] C. Bryce, M. Oriol, and J. Vitek. A coordination model for agents based on secure spaces. In P. Cinacrini and A. L. Wolf, editors, *Proc. of Coordination'99: Third International Conference on Coordination Models and Languages; LNCS 1594*, pages 4–20, April 1999.

[4] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[5] Paolo Ciancarini. Enacting rule-based software processes with polis. Technical report, University of Pisa, october 1991.

[6] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[7] R. De Niccola, G Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In David Garlan and Daniel Le Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 221–237, September 1997.

[8] E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In David Garlan and Daniel Le Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 274–288, September 1997.

[9] E. Denti and A. Omicini. An architecture for tuple-based coordination of multi-agent systems. *Software—Practice & Experience*, 29(12):1103–1121, 1999.

[10] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces(TM) Principles, Patterns and Practice (The Jini(TM) Technology Series)*. Addison-Wesley, 1999.

[11] D. Gelenter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[12] Jini. Technical report, Sun Microsystems. website: `http://java.sun.com/products/jini/`.

[13] P.F. Linington. Options for expressing ODP enterprise communities and their policies by using UML. In *Proceedings of the Third Internantional Enterprise Distributed Object Computing (EDOC99) Conference*. IEEE, September 1999.

[14] P.F. Linington, Z. Milosevic, and K. Raymond. Policies in communities: Extending the odb enterprise viewpoint. In *Proceedings of the Second Internantional Enterprise Distributed Object Computing (EDOC98) Conference*. IEEE, November 1998.

[15] J. J. Ch. Meyer, R. J. Wieringa, and Dignum F.P.M. The role of deontic logic in the specification of information systems. In J. Chomicki and G. Saake, editors, *Logic for Databases and Information Systems*. Kluwer, 1998.

[16] N.H. 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.

[17] N.H. Minsky and V. Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *7th USENIX Security Symposium*, January 1998.

[18] N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 2000. (to be published—available through `http://www.cs.rutgers.edu/~minsky/`).

[19] J. Pinakis. Providing directed communication in Linda. In *Proceedings of the 15th Australian Computer Science Conf.*, pages 731–743, 1992.

[20] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[21] M. Roscheisen and T. Winograd. A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[22] B. Schneier. *Applied Cryptography.* John Wiley and Sons, 1996.

[23] M. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 1994.

[24] S.W. Smith and S. H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, April 1999. (Special Issue on Computer Network Security).

[25] P. Tarau. Language issues and programming techniques in BinProlog. In *Proceedings of of the Gulp'93 Conference*, June 1993.

[26] V. Ungureanu. *A Mechanism for Supporting Communication Policies in Distributed Systems.* PhD thesis, Rutgers University, 2000. Obtainable from ungurean@cs.rutgers.edu.

[27] P. Wyckoff, W. McLaughry, T.J. Lehman, and D.A. Ford. TSpaces. *IBM Journal of Research and Development*, 37:454–474, 1988.

[28] B. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, May 1994.