Law-Governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems

Naftaly H. Minsky
and
Victoria Ungureanu
Rutgers University

Software technology is undergoing a transition from monolithic systems, constructed according to a single overall design, into conglomerates of semi-autonomous, heterogeneous and independently designed subsystems, constructed and managed by different organizations, with little, if any, knowledge of each other. Among the problems inherent in such conglomerates none is more serious than the difficulty to *control* the activities of the disparate agents operating in it, and the difficulty for such agents to *coordinate* their activities with each other.

We argue that the nature of coordination and control required for such systems calls for the following principles to be satisfied: (1) coordination policies need to be enforced; (2) the enforcement needs to be decentralized; and (3) coordination policies need to be formulated explicitly—rather than being implicit in the code of the agents involved—and they should be enforced by means of a generic, broad spectrum mechanism; and (4) it should be possible to deploy and enforce a policy incrementally, without exacting any cost from agents and activities not subject to it.

We describe a mechanism called law-governed interaction (LGI), currently implemented by the Moses toolkit, which has been designed to satisfy these principles. We show that LGI is at least as general as a conventional centralized coordination mechanism (CCM), and that it is more scalable, and generally more efficient than CCM.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; D.2.2 [**Software Engineering**]: Design Tools and Techniques; C.2.4 [**Computer Communication Networks**]: Distributed Systems; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces; J.7 [**Computer Applications**]: Computers in Other Systems—*Command and Control*

General Terms: Design, Performance, Security

Additional Key Words and Phrases: coordination of heterogeneous agents, policy enforcement, scalability

1. INTRODUCTION

Software technology is undergoing a transition from monolithic systems, constructed according to a single overall design, into conglomerates of semi-autonomous, heterogeneous and independently designed subsystems, constructed and managed by different organizations, with little, if any, knowledge of each other. Among the problems inherent in such conglomerates none is more serious than the difficulty to control the activities of the disparate agents operating in it, and the difficulty for open groups of such agents to *coordinate* their activities with each other.

Coordination, which can be defined as *"the managing of dependencies between agents in order to foster harmonious interaction between them"* [Malone and Crowston 1994], is indispensable for effective cooperation between autonomous agents, as well as for safe competition between them. A flock of birds, for example, must coordinate its flight in order to stay in formation; and car drivers must coordinate their passage through an intersection, if they are to survive this experience. Each such coordination involves a certain *policy*, i.e., a set of *rules of engagement*—such as stopping at a red light, in the driving case—that must be complied with by all participants for the activity in question to be safe and harmonious. This is true for coordination in software as well.

Consider, for example, a distributed database D, accessed by an heterogeneous set C of autonomous agents—the clients of D. in order to prevent denial of services to some clients due to overconsumption of database services by others, one may choose to employ the following policy [Malone et al. 1993]:

> A client x can present a query to the database only if it has a positive *budget* b_x available to it, and b_x is reduced by one for each such query. Budgets can be provided to individual clients by a designated *regulator*; and a client with a positive budget can give part of it to any of his peers.

Under this, to be called **budgeted consumption** (BC) policy, the `regulator` can *control* the system, limiting the total load on the database by the amount of budget it provides to clients; and the clients can *coordinate* their activities by exchanging units of budgets between them.[1]

The implementation of a coordination policy must ensure that the policy is actually observed by *every* agent subject to it. In the case of the BC policy, for example, one needs to ensure that no query is ever issued by a client who does not have a positive budget, and that budget of clients is changed only as specified. This is simple to do with a *close-knit group*, whose members can all be carefully constructed to satisfy the policy in question. This is how birds of a feather flock together, having their rules of engagement inborn. This is also how the processes spawned by a single program coordinate their activities, by having their rules of engagement built in by the programmer or by the compiler.

But the implementation of such policies is much more problematic with *open groups*, whose members are *heterogeneous*, and whose membership changes dynamically and might be large. Our view of the appropriate nature of such an enforcement mechanism is expressed by the set of guiding principles discussed below.

---

[1]As this example shows, *coordination* and *control* are closely related concepts, and we will often use them interchangeably.

First, the disparate members of an *open* group, which might have been built independently, have little reason to trust each other to observe any given policy—unless there is some enforcement mechanism that compels them all to do so. This observation suggests the following principle:

PRINCIPLE 1 (ENFORCEMENT). *A coordination policy for an open group needs to be enforced.*

The simplest way to enforce a coordination policy, and the one used often in current practice, is via a *central coordinator* that mediates between the interacting parties. For example, in the case of the BC policy above, one can place a single coordinator between the distributed database and its clients. This coordinator would maintain the budgets of all clients, not allowing any of them to violate this policy.

But such *centralized coordination mechanism* (CCM) is not scalable. When the size of group C grows, the centralized coordinator becomes a bottleneck, and a dangerous *single point of failure*—undermining the benefit of having the database D itself distributed. Moreover this lack of scalability is not really necessary in this case, since the BC policy is inherently local, in the sense that it can be complied with by each client, without any knowledge of the behavior of any other client. We believe that such local aspects of a policy should not require centralized enforcement. This leads to the following principle:

PRINCIPLE 2 (DECENTRALIZATION). *The enforcement mechanism should not require central control.*

Next, due to their *diversity*, conglomerate systems are likely to employ multitude of different coordination policies. And a single software agent operating within such a system may find itself interacting with several groups of agents, operating under disparate policies. If these policies are *implicit* in the code of the agents involved—as is most often the case in current practice—or if different policies are expressed by means of different formalisms and enforced with different mechanisms, it would be very difficult to deploy groups that must operate under a given policy. The need for *easy deployment* leads to the following, well known, principle:

PRINCIPLE 3 (SEPARATION OF POLICY FROM MECHANISM). *Coordination policies should be made **explicit**, and be enforced by means of a **single mechanism** that can implement a wide range of policies in a uniform manner.*

Finally, we note that in a large system, if deployment cannot be done incrementally, without making any requirements of the rest of the system, it probably cannot be done at all. This gives rise to the following principle:

PRINCIPLE 4 (INCREMENTAL DEPLOYMENT). *One should be able to deploy and enforce a policy* incrementally*, without exacting any cost from agents and activities not subject to it.*

We describe in this paper a coordination mechanism, called *law-governed interaction* (LGI), whose design has been based on these principles. The basic structure of LGI is discussed in Section 2, and is applied to our example BC policy, for illustration. Some additional features of LGI are introduced in Section 3, and illustrated with elaborations on the BC policy. The usage of LGI, its expressive power and

its limitations are discussed in Section 4. The theoretical efficiency of LGI, and the performance of its current implementation, via the Moses toolkit, are discussed in Section 5, where we show that LGI is generally more efficient, and more scalable, than centralized coordination. In Section 6 we review related work, and we conclude in Section 7.

2. LAW-GOVERNED INTERACTION (LGI)

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 section is organized as follows: We start in Section 2.1 by formally defining the concept of an $\mathcal{L}$-group. In Section 2.2 we present the other basic elements of LGI. The law-enforcement mechanism is discussed in Section 2.3. Our language for specifying laws is presented in Section 2.4, and its use is illustrated in Section 2.5, by formalizing the BC policy of the introduction. The deployment of $\mathcal{L}$-groups is discussed in Section 2.6. We conclude this section by listing the various components of the Moses toolkit that implements LGI.

2.1 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 $\{\mathcal{CS}_x \mid x \in \mathcal{A}\}$ 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 a certain type 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.

We distinguish between two kinds of $\mathcal{L}$-groups, regarding the management of their membership: *explicit* groups and *implicit* ones—both of which are supported by our Moses toolkit. An explicit $\mathcal{L}$-group $\mathcal{G}$ is one whose membership is maintained and regulated by a distinguished agent called the *secretary* of $\mathcal{G}$, to be denoted by $\mathcal{S}_{\mathcal{G}}$. This secretary maintains the law $\mathcal{L}_{\mathcal{G}}$, and the membership of $\mathcal{G}$—and it acts as a name-server for this group. Note that the secretary does not participate in the exchange of messages between members of the group, once they found each other. It thus has only a minor adverse effect on the scalability of the group.

An *implicit* $\mathcal{L}$-group assumes no knowledge of its total membership, and it does not use a central secretary. Anybody who operates under law $\mathcal{L}$ is considered a member of the implicit $\mathcal{L}$-group (there is, by definition, only one such group), and any two members of this group can exchange $\mathcal{L}$-messages, if they know each other's address. However, different members of an implicit group may be unaware of each other's existence, and there maybe nobody who knows what the membership of the group is.

We will focus on explicit groups throughout this paper, leaving implicit groups, whose implementation is now being tested, for a subsequent publication. We note however, that as far as the user is concerned, the main differences between these two types of groups is in their deployment, and in naming.

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. For example, under the law that implements the $\mathcal{BC}$ policy (defined in Section 2.5) each client will have in its control-state a term budget(b), where b would represent the value of the budget allocated to this client.

2.2 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 four types of regulated events. The first pair of events represents stages of the passing of an $\mathcal{L}$-message. The remaining two types of events, which will be discussed in detail only in Section 3, deal, respectively, with *obligations*, and with *exceptions*.

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 sent by x, arrives at $\mathcal{C}_h$. Ostensibly, since 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. (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 3.1).

4. exception(...)—the occurrence of this event means that some exceptional condition has been raised at its home. (Exceptions are discussed in Section 3.2).

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, 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.* These operations include the following, informally specified, ones:

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[2] t, if any; (3) t1←t2 which replaces term t1 with term t2 (it has no effect if t1 does not exist); (4) incr(t(v),d) which

---

[2]Note that a control-state is a *bag* of terms, so if there are two terms that match t, only one of them would be removed. Similar "bag-semantics" applies to other operations in this group.

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), carried out by $\mathcal{C}_x$, sends to controller $\mathcal{C}_y$ an $\mathcal{L}$-message m addressed to y—where x identifies the *nominal sender* of the message ("nominal" because x may have not been the sender of this message, or may have sent a different one—recall that this operation is generated by the law, which may have a reason to misrepresent reality). When a message thus forwarded to y arrives at $\mathcal{C}_y$, it would trigger an arrived(x,m,y) event at it. Note that if y is a *list* of agents, then a multicast is performed, in a similar manner.

   —Operation deliver(x,m,y) delivers the message m to the home-agent y—not to the controller of y— where x is the nominal sender of this message. Note that this operation is generally carried out by $\mathcal{C}_y$, the controller associated with y itself. But as we shall see in Section 5.2.4, any other controller can deliver to y, which is called "remote delivery".

3. **Operations on obligations:** There are two such operations (imposeObligation and repealObligation) which will be described in detail in Section 3.1:

2.3 The Law-Enforcement Mechanism

We start with two observations regarding the term "enforcement," as used here. First, we do not propose to coerce any agent to exchange $\mathcal{L}$-messages under any law $\mathcal{L}$—such an exchange is purely voluntary. The role of enforcement here is merely to ensure that any exchange of $\mathcal{L}$-messages, once undertaken, conforms to law $\mathcal{L}$. Yet, an agent may be *effectively compelled* to exchange $\mathcal{L}$-messages, and thus be subject to law $\mathcal{L}$, if he wishes to use services provided only under this law. For instance, if the database in our introductory example accepts only $\mathcal{BC}$-messages as queries, then anybody wishing to use this database would be compelled to send his queries under this law, and thus be subject to its budget restrictions.

Our second observation has to do with the condition under which conformance to a law is to be ensured. 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. In this paper we will be concerned with inadvertent violations—which is a typical software-engineering concern. Enforcement of coordination laws with respect to malicious violations—which is a security concern—is discussed in [Minsky and Ungureanu 1998b; Minsky and Ungureanu 1998a]; but we will make some comments about what such enforcement entails.

The rest of this section is organized as follows: we start by introducing controllers, which are our main enforcement tools, we then discuss the manner in which controllers mediate $\mathcal{L}$-messages, and we conclude with a discussion of the assurances provided by LGI.

The Controllers, and their Role. Broadly speaking, 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 1). 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 facilitate various optimization techniques, discussed in Section 5.

The Moses toolkit includes a *controller-server*, which maintains a set of active controllers. This server can provides the address of an available controller to anybody who wishes to engage in LGI. Alternatively, an agent may use a controller on its own host, if available.

Controller-mediated messages. Here is how the exchange of $\mathcal{L}$-messages gets to be mediated by controllers, and how this mediation is carried out. Consider a pair of agents x and y in $\mathcal{L}$-group $\mathcal{G}$, which have controllers $\mathcal{C}_x$ and $\mathcal{C}_y$, respectively, assigned to them. For x to send an $\mathcal{L}$-message m to y, it must send m to $\mathcal{C}_x$. Moses provides two interfaces for this purpose: (a) a class that allows a Java program to send and receive $\mathcal{L}$-messages, and (b) an interactive interface that allows a human agent to do the same from within a web-browser. When a message thus sent by x arrives at $\mathcal{C}_x$, it triggers a sent(x,m,y) event at it. When $\mathcal{C}_x$ picks up this event, it evaluates the ruling of law $\mathcal{L}$ for it with respect to control-state $\mathcal{CS}_x$ that it maintains, and carries out this ruling.

If, in particular, the ruling calls for the control-state $\mathcal{CS}_x$ to be updated, such update is carried out directly by $\mathcal{C}_x$. And if the ruling for event sent(x,m,y) calls for message m to be forwarded to y, then $\mathcal{C}_x$ would forward m to the controller $\mathcal{C}_y$ instead. When this message arrives at $\mathcal{C}_y$ it triggers an arrived(x,m,y) event at it. Controller $\mathcal{C}_y$ would then evaluate and carry out the ruling of the law for this event. This ruling, in turn, might, in particular, call for m to be delivered to y, and for the control-state $\mathcal{CS}_y$ maintained by $\mathcal{C}_y$ 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, with priority is given to obligationDue events (to be discussed later). 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.

Assurances. For this enforcement scheme to be effective one needs the following assurances: (a) that the exchange of $\mathcal{L}$-messages is mediated by controllers interpreting the *same law* $\mathcal{L}$; and (b) that all these controllers are *correctly implemented*. If these two conditions are satisfied, then it immediately follows that if y receives an $\mathcal{L}$-message from some x, this message must have been sent as an $\mathcal{L}$-message. In other words, one cannot forge $\mathcal{L}$-messages.
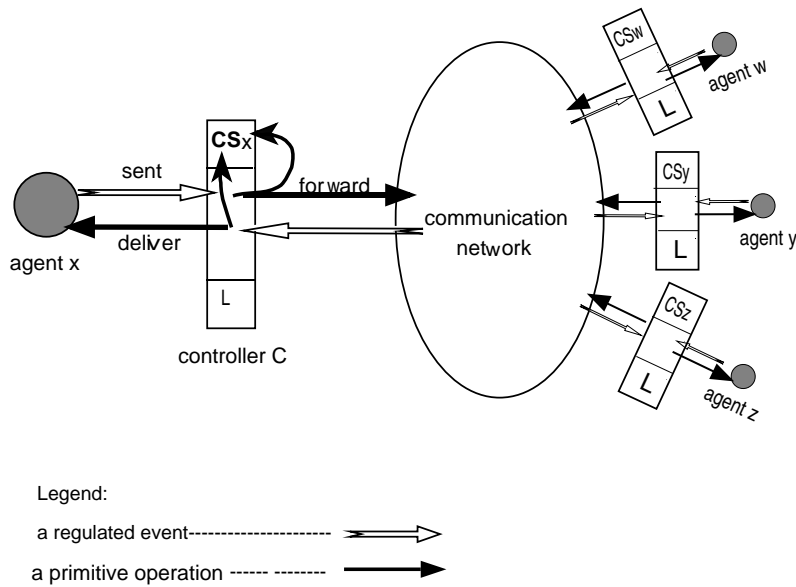
Fig. 1.   Enforcement of the law

Regarding the first of these concerns, to ensure that a message forwarded by controller $\mathcal{C}_x$ under law $\mathcal{L}$ would be handled by $\mathcal{C}_y$ under the *same* law, $\mathcal{C}_x$ appends a hash H of law $\mathcal{L}$ to the message it forwards to $\mathcal{C}_y$. (The hash of the law is obtained using one way functions which transforms any string into a considerably smaller bits sequence with high probability that two strings will not collide [Rivest 1992; Schneier 1996].) Controller $\mathcal{C}_y$ would accept this as a valid $\mathcal{L}$-message only if H is identical to the hash of its own law.

Let us turn now to the second concern, about the correctness of the controllers: When not concerned with malicious violations, then one can trust a controller provided by our controller-server, or a controller provided by the operating system— just like we often trusts various standard tools on the Internet, such as the e-mail software or browsers. When malicious violations are a concern, the validity of controllers, and of the host on which they operate needs to be certified, and the controller-server mentioned above needs to operate as a *certifying authority* for controllers. Also, in this case, messages sent across the network must be digitally signed by the sending controller, and the signature must be verified by the receiving controller—allowing the two controllers to trust each other. Such secure inter-controller interaction has been implemented in Moses ([Minsky and Ungureanu 1998a]).

2.4 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 [Clocksin and Mellish 1981], due to its expressive power, and its relatively widespread usage. (The restrictions on the Prolog used here include that we do not permit such goals as `asserta`, `retract` and `call`.)

This language will be introduced briefly in this section, and used in the rest of this paper. We are aware of the drawbacks of this choice, which include longer time for law-enforcement, and difficulties in reasoning about laws. We plan to replace Prolog with a simpler and more restrictive language, when this will become necessary for an application domain. Such a change will be easy to accomplish, because it requires no other change in the LGI model, and only a minor change in the Moses toolkit that implements it.

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 R of primitive-operations representing the ruling of the law for this event. (We assume that program L *terminates*, for every possible event. Since there are no automatic means to guarantee termination of Prolog programs, we do need to rely on the designer of the law to ensure termination.)

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 term t with each term in the control-state of the home agent. For example, under law $\mathcal{BC}$, the execution of goal budget(B)@CS would match the budget term in the home control-state, binding variable B to the budget. A *do-goal* has the form do(p), where p is one of the above mentioned primitive-operations. It appends term p to the ruling of the law. For example, the goal do(+budget(100)) would add the specified primitive to the ruling.

The ruling of the law is computed as follows. The interpreter of the law maintains an auxiliary variable R that starts, at the beginning of the evaluation, as an empty list, and whose value at the conclusion of the evaluation would become the ruling of law $\mathcal{L}$ for the given event e. The list R is constructed by means of do-goals, as follows: a do-goal do(p) succeeds if the term "p" is bound to a valid form of a primitive-operation; if it succeeds then the term p is appended to the list R, as a tentative contribution of the ruling of the law (tentative, because this contribution is retractable upon backtracking.) More details about this process is provided by [Minsky 1991]. We conclude this discussion with several technical details and conventions, and will then proceed with an example.

Evaluation Context. Just before presenting law L with an event e for evaluation, the controller instantiates the following Prolog variables, which provide the context in which the ruling is evaluated.

CS:. The control state of the home object, represented as a list of terms (in unspecified order).

Msg:. The message being sent or received.

Self:. The name of the home agent.

Peer:. The peer for the current event: the recipient in the case of sent-event, and the sender in the case of the arrived-event.

Clock:. The local time at the home controller.

Abbreviations for the Primitives `forward` *and* `deliver:`. The primitive operation `forward(x,m,y)` is often abbreviated into `forward`, which is syntactic sugar for `forward(Self,Msg,Peer)`. Similarly, operation `deliver(x,m,y)` is often abbreviated into `deliver`, which means `deliver(Peer,Msg,Self)`.

2.5 The Budgeted Consumption ($\mathcal{BC}$) Law—an Example

We now show how the *budgeted consumption* policy introduced informally in Section 1 can be formalized into a law, and implemented under LGI. Law $\mathcal{BC}$ in question is specified in Figure 2. This figure starts with the description of the initial control-state of the various agents. 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. (The setting of the initial control-state of agents is discussed in Section 2.6.)
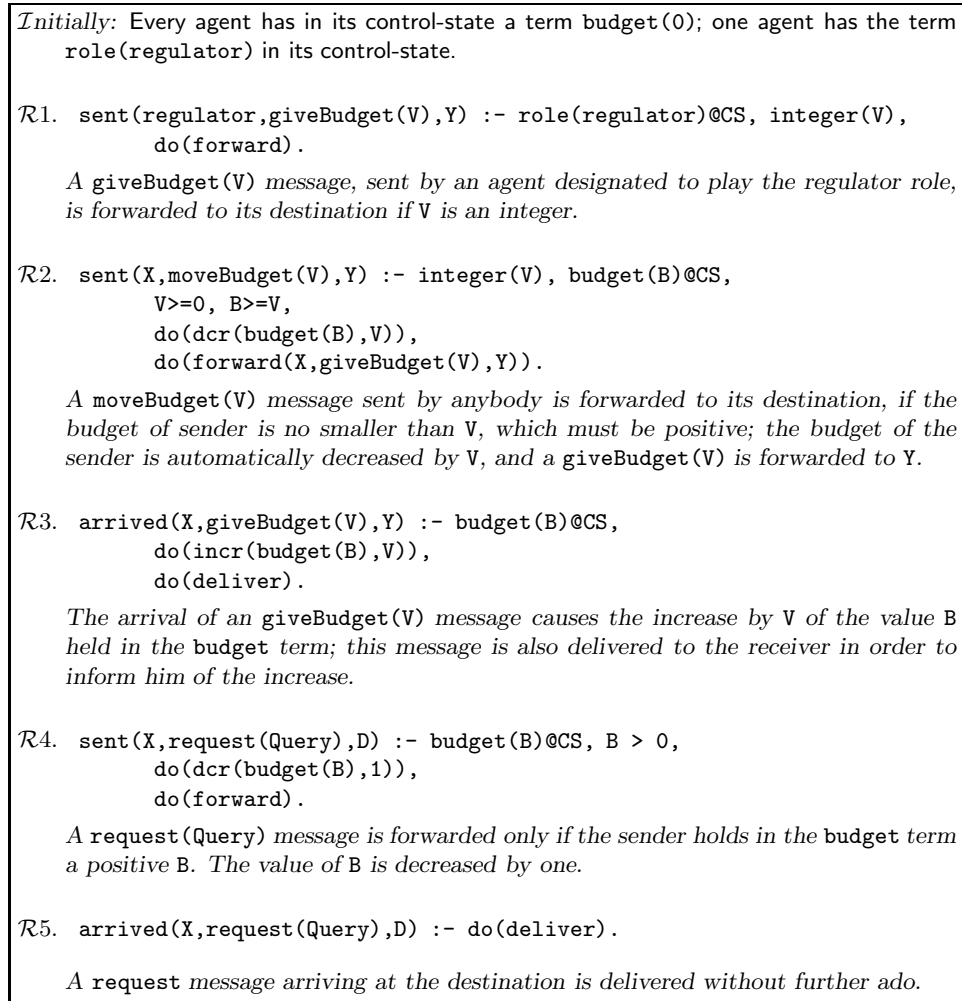
Under law $\mathcal{BC}$, the budget b_x of agent x is represented by the value b of the term `budget(b)` in the control-state of x. These budgets are initially zero, for every agent in the group. We start our discussion of this law by showing how budgets can be allocated and distributed under it. We will then show how database queries are regulated by means of these budgets.

The budget of an agent under $\mathcal{BC}$ can change in several ways. First, an agent designated as a *regulator* (by a term `role(regulator)` in its control-state) can change the budget of any agent y by an arbitrary integer value v, simply by sending a message `giveBudget(v)` to y. The sending of such a message is authorized by Rule $\mathcal{R}1$, which forwards the message to its destination. By Rule $\mathcal{R}3$, when this message arrives at y, the budget of y would be incremented by v (it would *decrease* if v is negative).

Second, an agent x that has a positive budget in its control state can *move* part of it to another agent y. This it can do by sending the message `moveBudget(v)` to y. The sending of this message is authorized by Rules $\mathcal{R}2$, which reduces the budget of x by v units, and then forwards to y the message `giveBudget(v)` (note how the law can change a message in flight.) The arrival of this message is (as we have seen before) regulated by Rule $\mathcal{R}3$, would cause the budget of y to increase by v.

Thus, it is the regulator that controls the total amount of budget available for database queries (note that due to the initial state of this group, it has only one regulator). But the regulator does not need to allocate the budget of every individual client. It might provide budgets only to a small group of "managers," expecting each of them to distribute their budget to their staff, as they see fit. The budget distribution strategy is not addressed by this law, and is left for the regulator, and for other agents, to determine. We will return to this point in Section 4.

We turn now to the effect that budgets have on database queries: By Rule $\mathcal{R}4$, a message `request(query)` sent by an agent x with budget b, will be forwarded to its destination only if b is positive; and b would be automatically decremented by 1 when this query is forwarded. When the query arrives at a database server, it is delivered without further ado (Rule $\mathcal{R}5$). A server is expected to respond to this request with an appropriate reply, which is not regulated under this law. (Note

Initially: Every agent has in its control-state a term budget(0); one agent has the term role(regulator) in its control-state.

$\mathcal{R}$1. ```sent(regulator,giveBudget(V),Y) :- role(regulator)@CS, integer(V),
        do(forward).```

   A giveBudget(V) *message, sent by an agent designated to play the regulator role, is forwarded to its destination if* V *is an integer.*

$\mathcal{R}$2. ```sent(X,moveBudget(V),Y) :- integer(V), budget(B)@CS,
        V>=0, B>=V,
        do(dcr(budget(B),V)),
        do(forward(X,giveBudget(V),Y)).```

   A moveBudget(V) *message sent by anybody is forwarded to its destination, if the budget of sender is no smaller than* V*, which must be positive; the budget of the sender is automatically decreased by* V*, and a* giveBudget(V) *is forwarded to* Y*.*

$\mathcal{R}$3. ```arrived(X,giveBudget(V),Y) :- budget(B)@CS,
        do(incr(budget(B),V)),
        do(deliver).```

   The arrival of an giveBudget(V) *message causes the increase by* V *of the value* B *held in the* budget *term; this message is also delivered to the receiver in order to inform him of the increase.*

$\mathcal{R}$4. ```sent(X,request(Query),D) :- budget(B)@CS, B > 0,
        do(dcr(budget(B),1)),
        do(forward).```

   A request(Query) *message is forwarded only if the sender holds in the* budget *term a positive* B*. The value of* B *is decreased by one.*

$\mathcal{R}$5. ```arrived(X,request(Query),D) :- do(deliver).```

   A request *message arriving at the destination is delivered without further ado.*

Fig. 2. The Budgeted Consumption Law $\mathcal{BC}$

that this law does not actually ensure that a query can be sent and delivered only to a database server. This would be easy to do, but seems not to be needed here, because clients are not likely to send queries to other agents, thus losing part of their budget without any return.)

In accordance with our principles of coordination in open groups, the $\mathcal{BC}$ policy is *enforced*, and does not rely on the voluntary compliance of individual agents. Moreover, $\mathcal{BC}$ is enforced *locally*, at each agent, and is therefore quite scalable. We will elaborate on this example in Section 3, making it more robust and fault tolerant, using additional features of LGI.

2.6 The Deployment of $\mathcal{L}$-Groups

A new $\mathcal{L}$-group $\mathcal{G}$ is established by starting up a secretary $\mathcal{S}_{\mathcal{G}}$ which maintains: (a) the law $\mathcal{L}$ of $\mathcal{G}$; (b) the list of its initial members, with their initial control-states;
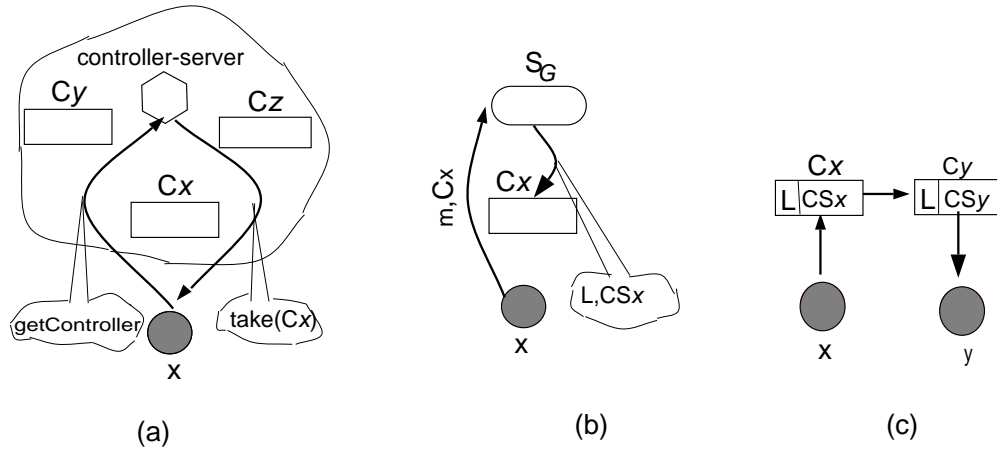
Fig. 3. Deployment of an $\mathcal{L}$-group: (a) agent x asks the controller-server for a controller—it gets C_x in this case; (b) agent x asks $\mathcal{S}_\mathcal{G}$, the secretary of group $\mathcal{G}$, to become an active member—causing C_x to be loaded with $\mathcal{L}$, the law of group $\mathcal{G}$, and the control-state of x; (c) x can start communicating with another member y operating under the same law.

and (c) the initial control-state to be assigned to new members that might join the group during its lifetime. In the case of $\mathcal{BC}$-group, in particular, one may have only a single initial member, whose initial control-state would consist of the term `role(regulator)`, and the control-state of new members would consist of the term `budget(0)`. (This setting is summarized in the "Initially" clause of Figure 2.)

For an agent x to be able to exchange $\mathcal{L}$-messages under a law $\mathcal{L}$, it needs to engage in a two-step connection protocol, illustrated by Figure 3. First, x must employ a generic controller $\mathcal{C}_x$, that knows nothing about any particular policy. This may be done by sending a message to an agent called *controller-server* that operates as a trusted name-server for controllers. Alternatively, x may have a controller available on its own host, or at its own LAN.

The second step is to join the group by sending a `connect` request to the secretary $\mathcal{S}_\mathcal{G}$. In this request x must identify its controller $\mathcal{C}_x$, and the name m of the member it wants to animate (i.e., to operate as the agent of). The secretary might attempt to authenticate agent x (via a password) and its chosen controller (via a certificate), if authentication is required for this particular group. Once this is done, the secretary would proceed as follows: (a) it would provide $\mathcal{C}_x$ with law $\mathcal{L}$; and (b) it would send $\mathcal{C}_x$ the initial control-state $\mathcal{CS}_x$ to be assigned to x.

This control-state is determined as follows: If x asks to animate one the initial members of the group, it will get the designated initial control-state of this member. For example, if x asks to animate the regulator, of our $\mathcal{BC}$-group, it will get the term `role(regulator)` in its control-state. If, on the other hand, x asked to animate a new member of this group, it will get the control-state designated by the secretary for new members—which, for our $\mathcal{BC}$-group, consists of the term `budget(0)`.

Once x is thus engaged in an $\mathcal{L}$-group, it can exchange $\mathcal{L}$-messages with other members of this group, whose name and address is provided by the secretary; such

an exchange is mediated by the controllers of each member, and does not involve the secretary $\mathcal{S}_\mathcal{G}$. To carry out such an exchange, an agents needs to the interfaces provided by Moses for this purpose (see the following section), so it must be aware of its participation in an $\mathcal{L}$-group interaction. Note, however, that we are working now on allowing the kernel of an operating system to intercept regular messages, subjecting some of them to the ruling of a specified law law $\mathcal{L}$. The main purpose of such interception would be to subject certain COTS (commercial of the shelf) components to certain laws.

Note also that a member of an $\mathcal{L}$-group does not have to be familiar with the details of law $\mathcal{L}$ itself, although it might need to be aware of certain aspects of this law, to operate effectively. In the case of an $\mathcal{BC}$-group, for example the clients need to understand the law-based concept of budget, in order to plan their actions.

2.7 The Moses Toolkit

The Moses toolkit consists of the following components, written mostly in Java:

—The *controller*.
—Two kinds of *interfaces* between agents and controllers: one for a human agents, operating via a web-browser, and one for Java programs.
—The *secretary*, which is needed only for explicit $\mathcal{L}$-groups.
—A *controller server* that maintains information about active controllers, throughout the Internet. (One may have any number of such servers.)

3. ADDITIONAL FEATURES OF LGI

In this section we introduce two additional features of LGI, which enhance its expressive power and its usefulness considerably. The first is a concept of obligation, which allows us to write laws that ensure liveness properties, among other things. The second feature is a concept of exception, which helps to write more robust and fault tolerant laws. Note that LGI has some other important features, mention briefly in the conclusion, which are beyond the scope of this paper.

3.1 The Concept of enforced obligation

So far, our concept of law has been purely *reactive*. That is, it prescribes what should be done in response to `sent` and `arrived` events, but it cannot initiate any action on its own. To see the need for a more pro-active role for laws, consider the following problem with our $\mathcal{BC}$ law: an agent x invested with a given budget might actually not need it, or might be busy with other things for a while—so that the budget of x is left unused for a long period of time, while other agents might be starving for some query-budget. To help utilize such unused budgets, one may want to employ the following amendment to policy $\mathcal{BC}$:

> If a client x does not issue any query for a certain period of time (say, 10 seconds) it must give up its query budget, by sending it to the regulator. (Thus amended, our policy is called $\mathcal{BC}'$.)

One can ensure that unused budgets are returned in time, as is required by this policy, by means of the concept of *enforced obligation* (or, *obligation*, for short) presented below:

Informally speaking, an obligation incurred by a given agent serves as a kind of *motive force*, which ensures that a certain action (called *sanction*) is carried out at this agent, at a specified time in the future (the deadline) 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.

Specifically, suppose that at time t_0, an agent x incurs an obligation by the execution at x of a primitive operation

<div align="center">

`imposeObligation(oType,dt)`

</div>

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*

<div align="center">

`obligationDue(oType)`

</div>

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

<div align="center">

`repealObligation(oType)`

</div>

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.

The $\mathcal{BC}'$ Law—Illustrating the Use of Obligations. Our amendment to the $\mathcal{BC}$ law is implemented by replacing Rules $\mathcal{R}3$ and $\mathcal{R}4$ from Figure 2 with Rules $\mathcal{R}3'$ and $\mathcal{R}4'$, and by adding Rule $\mathcal{R}6$. These rules, displayed in Figure 4, operate as follows.

By Rule $\mathcal{R}3'$, whenever x receives an `giveBudget` message, an obligation `returnBudget` is imposed on it, with a deadline of 10 seconds. The general effect of this obligation is that if during a period of 10 seconds this agent made no queries then its budget would be taken away from it. When the agent does make a query then this obligation would be replaced with a new one that would come due, again, in 10 seconds.

More specifically, if this obligation will come due—i.e., if it is not repealed, as under Rule $\mathcal{R}4'$—its sanction (by Rule $\mathcal{R}6$) would be to forward to the `regulator` a `giveBudget(b)` message, where `b` is the value of x's budget, and to change the value of x's `budget` to 0. In other words, x is forced to give away its budget to the regulator. (We assume here, for simplicity, that the agent which plays the role of regulator is also called "regulator").

Second, by Rule $\mathcal{R}4'$, if an agent x issues a request for one of the database servers, and if it has a positive balance in its budget, then its pending `returnBudget` obligations are repealed (an agent might have several such obligations, because

it incurs one each time it receive some budget), because x thus complies with obligation to use his budget within the alloted time period. Second, to ensure that x will observe our law in the next time frame, a new obligation returnBudget is imposed, to come due in 10 seconds. Finally, the budget is decremented and the query is forwarded, as before.
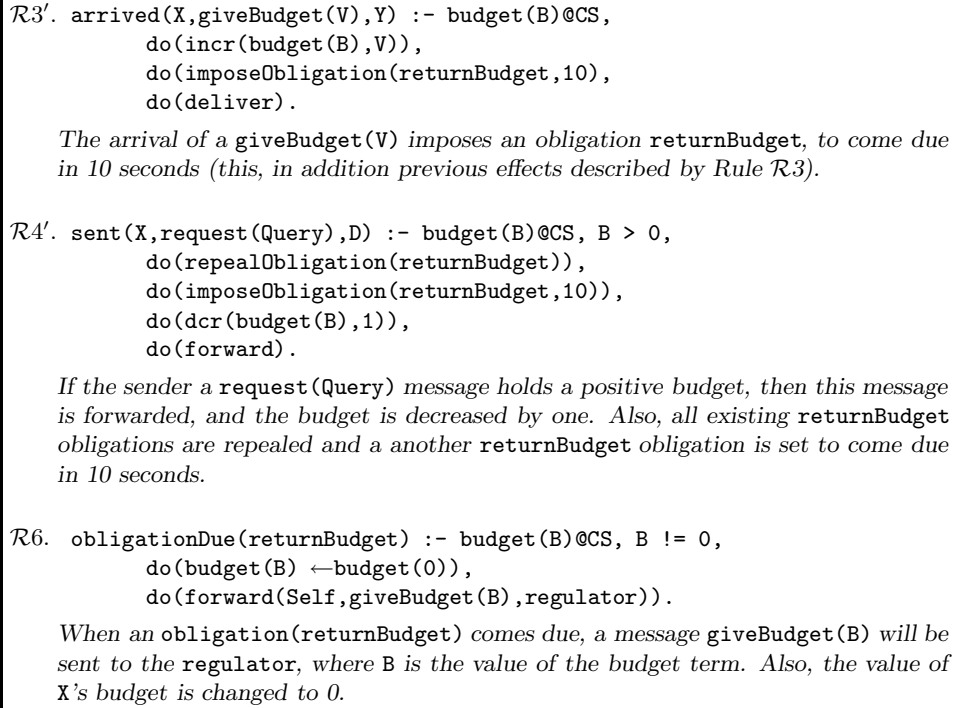
```
R3'. arrived(X,giveBudget(V),Y) :- budget(B)@CS,
          do(incr(budget(B),V)),
          do(imposeObligation(returnBudget,10),
          do(deliver).
```
The arrival of a giveBudget(V) *imposes an obligation* returnBudget, *to come due in 10 seconds (this, in addition previous effects described by Rule R3).*

```
R4'. sent(X,request(Query),D) :- budget(B)@CS, B > 0,
          do(repealObligation(returnBudget)),
          do(imposeObligation(returnBudget,10)),
          do(dcr(budget(B),1)),
          do(forward).
```
If the sender a request(Query) *message holds a positive budget, then this message is forwarded, and the budget is decreased by one. Also, all existing* returnBudget *obligations are repealed and a another* returnBudget *obligation is set to come due in 10 seconds.*

```
R6. obligationDue(returnBudget) :- budget(B)@CS, B != 0,
          do(budget(B) ←budget(0)),
          do(forward(Self,giveBudget(B),regulator)).
```
When an obligation(returnBudget) *comes due, a message* giveBudget(B) *will be sent to the* regulator, *where* B *is the value of the budget term. Also, the value of* X*'s budget is changed to 0.*

Fig. 4.   Rules of law $\mathcal{BC}'$ dealing with unused budget

Obligation in Deontic Logic. Conventional *deontic logic* [Meyer et al. 1998; Brown 1998; Kent et al. 1993; S. 1993], designed for the specification of normative systems, is based on a related concept of obligation. The deontic concept of obligation allows one to *reason* about what an agent must do, but it provides no means for ensuring that what needs to be done will actually be done [Roscheisen and Winograd 1996]. Such obligations, have been used, in particular, for the specification of policies for financial enterprises [Linington 1999]—but unlike our obligations, they provide no direct help in the enforcement of such policies.

Further Details of Obligations. The following details provide a completion of our concept of obligation, which may be skipped in first reading.

A secondary effect of an imposeObligation(oType,dt) operation is that the term obligation(oType,t_0,dt) is added to $\mathcal{CS}_x$, where t_0 is the time when this operation has been executed. This term is used as the indication that agent x has a *pending obligation* of the specified type and deadline, and it is removed

automatically when the associated `obligationDue` event occurs. It is also removed by the execution of any `repealObligation(oType)` operation. But an `obligation` term cannot be added, removed or changed directly by update operations on the control-state.

The result of all this is the following invariant: an obligation of type `oType` on agent `x`, which is set at time t_0 to fire in `dt` seconds, is always accompanied by the term `obligation(oType,`t_0`,dt)` in the control-state of `x`. This property provides the law with the very important ability to "introspect" on the current set of pending obligations, by examining the `obligation`-terms.

Another role of `obligation`-terms is to specify *initial obligations* of agents. This is done as follows: A term `obligation(oType,0,dt)` in the *initial* control-state of an agent `x` means that an `obligationDue(oType)` event is to be triggered at `x`, `dt` seconds after `x` started operating. Accordingly, the following actions will take place if an agent `x`, having in its initial control state a term `obligation(oType,0,dt)`, is launched at time t_0: (a) the term `obligation(oType,0,dt)` is removed, and (b) primitive `imposeObligation(oType,dt)` is executed at time t_0. For the use of this capability see the token-ring example in [Minsky et al. 1996].

Finally, we offer the following details about the execution of incurred obligation:

—Events are evaluated in chronological order (of arrival times); in case of a tie, the evaluation of an `obligationDue` event takes precedence over other types of events. If multiple obligations come due at the same time, they are evaluated in some unspecified order.

—When an `obligationDue` event occurs, no values are bound to `Msg` or `Peer`, for obvious reasons; however, `Self` is defined as always.

3.2 Exceptions

Primitive operations that require communication with other agents, like `deliver` and `forward`, may end up not being able to fulfill their intended function. For example, the destination agent of a `forward` operation may fail by the time the forwarded message arrives at it. To deal with this problem, we introduce a regulated event called an `exception`, which is triggered when a primitive operation cannot be completed successfully—and it is up to the law to prescribe what should be done to recover from such an exception. The syntax of an `exception` event is:

$$\text{exception(primitiveOperation, failureCause)}$$

where `primitiveOperation` is the primitive operation that could not be completed, and `failureCause` is a string describing the reason of the failure. The home of the `exception` event is the home of the event which attempted to carry out the failed `primitiveOperation`. For instance, if a message `m`, sent on behalf of member `x`, cannot be delivered to its destination `y`, then an event

$$\text{exception(deliver(x,m,y),''destination not responding'')}$$

would be triggered at `x`.

The $\mathcal{BC}''$ Law—Illustrating the Use of Exceptions. Consider again the issue of fair distribution of budgets among clients of the database. The previous amendment of law $\mathcal{BC}$ ensured that only agents constantly using the database keep their budgets.

Another aspect of this issue is to guarantee that quantities of budget are not lost in the distribution process. Consider an agent x who sent a `giveBudget(v)` message to another agent y, and suppose that y's controller, or the network, failed and the message cannot be forwarded. Then v units of budget would be lost, without a trace. What one would like in this case is to ensure the following property:

> If a `giveBudget` message cannot be forwarded, then the sender's budget is restored.

This can be accomplished by adding Rule $\mathcal{R}7$, presented in Figure 5, to law $\mathcal{BC}'$, thus establishing a variant $\mathcal{BC}''$ of our law. This rule deals with the exception that is triggered when the `giveBudget(v)` message sent by an agent x cannot be forwarded to y. The ruling for this exception is to restore x's budget by increasing it by v; and to notify x by delivering an appropriate memo to it.

```
R7.  exception(forward(X,giveBudget(V),Y)),FailureCause) :- budget(B)@CS,
          do(incr(budget(B),V)),
          do(deliver(memo(FailureCause))).
```

If an exception is raised because a `giveBudget` message sent by an agent X cannot be forwarded to Y, then X's budget is restored, and an appropriate memo is delivered to X.

Fig. 5.   Dealing with Lost Budget

4. THE USAGE OF LGI, AND ITS EXPRESSIVE POWER

Research on coordination was largely motivated by the conviction that when dealing with a distributed or parallel system, it is useful to distinguish between the computation carried out by individual components, and the communication between these components, which is sometimes called the *coordination protocol* [Carriero and Gelernter 1992]. Our concept of law is related to such protocols, but not identical to them. The law under LGI is not intended to specify *all* the details of the coordination between the members of a given group—it is merely a constraint over such a protocol.

For example, the law of a $\mathcal{BC}$-group defines the means for transferring quantities of budgets between agents, which ensures conservation of the total budget provided by the regulator—leaving it up to individuals agents to decide when to carry out such a transfer, and to whom.

It is not always clear which aspect of a given coordination protocol should be formulated as a law, and which should be left for individual agents. But as a general rule, any policy that is critical for the coordination, and which can be violated by several different agents, needs to be ensured by means of a law. In the case of the $\mathcal{BC}$-group, for example, the meaning of budgets, as currency for interaction with the database, and the conservation of such budgets when moved from one agent to another, are formulated by law $\mathcal{BC}$ because they can be reasonably considered critical, and because they can be violated by any agent if not ensured by a law.

Of course, not every policy deemed to be critical can be thus assured by a law, because LGI deals only with the exchange of message between agents, and is not

sensitive to the internal behavior of agents, and to changes in their internal state. For a sense of the kind of policies that can be usefully made into laws under LGI, we provide at the end of this section a brief overview of the applications of LGI that we already explored.

In general, LGI is most effective for policies that are naturally *local*. That is, policies that deal with the individual behavior of agent, requiring little or no knowledge of the state of other agents. It is for such policies that LGI is truly scalable, and most efficient (as discussed in Section 5.1). But LGI can be used for non-local policies as well. Detailed examples of implementation of such policies under LGI include *confidential electronic voting* [Minsky and Ungureanu 1997], and access-control for tuple-spaces [Minsky et al. 2000b]. In fact, as we show next, the set of policies that can be established by means of laws under LGI is at least as large as the set of policies that can be established using a centralized control mechanism (CCM)—and the efficiency of the LGI implementation is comparable to, or better than, the CCM implementation.

Comparing the expressive power of LGI to that of CCM. A centralized coordination mechanism (CCM) operates, essentially, as follows: Consider a set S of *participants* in a certain interactive activity that needs to be coordinated. Suppose that all messages sent by one participant to another are (somehow) intercepted, and rerouted to a *coordinator* C, for disposition. The response of the coordinator regarding each such message can be sensitive to the history of participants activities, as far as this is known to C from the intercepted messages.

Such a mechanism can be implemented under LGI as follows: Let the members of a group $\mathcal{G}$ consists of the set of participants S, and an agent C' whose role here is analogous to that of C, under CCM. For any policy P that can be implemented under CCM above, we can write a law $\mathcal{L}$ such that: (a) every $\mathcal{L}$-message sent by a participant is rerouted to C': (b) every bit of information about the history of the activities of members of S, maintained by the coordinator C, is also maintained by C'; and (c) the ruling of law $\mathcal{L}$, carried out by C' for a message thus arriving from a participant, is identical to the ruling of P carried out by C. (Our ability to do so depends only on the expressive power of the language used to write laws, which is virtually unlimited in the case of Prolog).

So, any policy that can be implemented under CCM can also be implemented under LGI. The opposite is not quite true. What cannot be regulated under CCM is the flow of messages sent by participants, and rerouted to the coordinator. This means, in particular, that CCM cannot protect the coordinator itself against congestion due to some overactive participants which may lead to denial of service (the perilous effects of such congestion has been described in [Needham 1994]). Under LGI, on the other hand, the law may limit the frequency of messages that can be issued by any given participant—and this limit can be locally enforced, and is, thus, less susceptible to congestion.

A sample of application of LGI. To provide some feel for the range of policies that can be practically established under LGI, we mention here some applications of LGI that have been studied, and most of them published elsewhere.

First, we experimented with many *coordination* laws, including a sophisticated *token-ring protocol* that ensure liveness of the token [Minsky et al. 1996], and a law

that provides for *confidential electronic voting* [Minsky and Ungureanu 1997].

Second, we addressed the issue of *dynamic reconfiguration* of distributed systems, while they are running, with the following result: Given a class of reconfigurations $\mathcal{R}$, deemed to be potentially important for a given system, it is sometimes possible to write a law that provides a suite of primitive *mutations* of the system that can be used to carry out any of the reconfigurations in $\mathcal{R}$, in an ad-hoc but safe manner [Minsky et al. 1996].

Third, we addressed the issue of *access control* [Minsky and Ungureanu 1998b]. We demonstrated that LGI can be used to support efficiently, and in a *unified* manner, a wide range of access control policies, including: conventional *discretionary* policies, *mandatory* lattice-based access control policies, and the more sophisticated policies required for commercial applications, such as the "Chinese Wall" policy [Brewer and Nash 1989], in distributed context, which is notoriously difficult to implement by traditional means. Another use of LGI has been as an access control mechanism for Linda-like tuple-spaces, where conventional access control schemes are inadequate [Minsky et al. 2000b; Minsky et al. 2000a].

Finally, we began exploring the emerging field of electronic commerce [Minsky and Ungureanu 1998a]. A commercial policy can be viewed as the embodiment of a contract between the principals involved in a certain type of commercial activity, and it may be concerned with such issues as: ensuring that a payment for services is refunded under specified circumstances; preventing certificates representing e-cash from being duplicated; ensuring that credit card numbers are used only for the transaction they are intended for; and, for certain socially sensitive transactions like the purchase of drugs, ensuring auditability by proper authorities. We have experimented with examples of all such policies. Our latest work in this field uses LGI to regulate agent involvement in inter-enterprise commerce [Gal et al. 1999].

5. ON THE EFFICIENCY OF LGI

As a gauge for the efficiency of LGI we will use the *relative overhead* $ro_{x,y}$ of transmitting an LGI message from x to y. We define this quantity to be the overhead incurred when transmitting an LGI message (i.e., the difference between the transfer time of the LGI message and the corresponding *unregulated* (TCP/IP, say) message), divided by the transfer time of the unregulated message.

We will evaluate the relative overhead (ro) of LGI under various conditions, comparing these results with the corresponding relative overhead of message passing under conventional centralized coordination mechanisms (CCM).

The general picture that emerges from this section is as follows: LGI communication should be quite affordable, and is generally more efficient than CCM communication—dramatically more efficient, in most cases. To be more specific: For communication across WAN (wide area network), the relative overhead (ro) of LGI is expected to be around 0.02 in most cases, which reflects quite a negligible cost. For comparison, the ro for CCM, is generally around 1, and could grow much higher when the central coordinator becomes congested. LGI is less efficient when security is at stake, but it is still comparable with CCM.

As for communication within a LAN (local area network), the expected relative overhead of LGI is between 0.2 and 0.4, depending on circumstances—significantly higher than over a WAN. But this should be compared with centralized coordination

within LAN, whose *ro* is estimated to be 1.4—three times as much as under LGI, and subject to congestion, to boot.

We start this section with an analysis of the structure of relative overhead of LGI, and of CCM; we also discuss the scalability of both mechanisms, when the frequency of messages exchanged between members of a group grows. In Section 5.2 we evaluate the relative overhead under various conditions, mostly focusing on WAN communication. Finally, in Section 5.3, we report on the testing of the performance (efficiency and robustness) of our current prototype implementation of Moses.

5.1 An Analysis of the Relative Overhead of LGI

Consider a message m sent by an agent x to a destination y. If the interaction between the two agents is mediated by controllers in the manner described in Section 2.3, then this message would be converted to three consecutive messages: (1) from x to $\mathcal{C}_x$, (2) from $\mathcal{C}_x$ to $\mathcal{C}_y$, and (3) from $\mathcal{C}_y$ to y. The overhead $o_{x,y}$, due to the extra messages and the law-evaluations involved, is given by the following formula:

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

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. (Note that some components of this formula can be eliminated under certain circumstances, as we shall see below.) The *relative overhead* $ro_{x,y}$ of an LGI message from x to y—relative to the unregulated transmission of such a message—is defined as:

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

For comparison, the relative overhead under centralized coordination (CCM), is given by the following formula:

$$ro_{x,y}^{CC} = o_{x,y}^{CC}/t_{com}^{x,y} = ((t_{com}^{x,C} + t_{eval}^C + t_{com}^{C,y}) - t_{com}^{x,y})/t_{com}^{x,y} \tag{3}$$

where the superscript C stands for the central coordinator under CCM, and the superscript CC denotes CCM itself.

Typical values. When evaluating these formulae under various conditions we will use the following approximations, and typical values, for the quantities involved in them.

—**Typical communication times** $t_{com}^{a,b}$. These times depend on many factors, including the length of message, the communication protocol being used, and the distance between the communicating parties. We will ignore many of these factors, and distinguish only between the following three 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_{LAN} (≈ 5ms): the TCP/IP communication time, for for a and b that reside in the same LAN.

3. t_{WAN} ($\approx$ 100ms): the TCP/IP communication time, for for a and b that reside in different LANs.

—**Typical evaluation times** t_{eval}^e**.** We ignore dependency on the event e, and on the law, and will use only two values:

1. t_{eval} ($\approx$ 1ms): the evaluation time for an arbitrary event, by a controller dedicated to the home of this event. (Our current, experimental, controllers, the evaluation time is 3.5ms; the figure of 1 ms used here is based on a very conservative expectation, explained in Section 5.3, about a performance of tighter controller, built with a more efficient Java.)

2. t_{eval}^C ($\approx$ 2ms): the expected evaluation time by an uncongested central-coordinator, which we take to be twice t_{eval}. (This is the case for a Moses controller, when it is shared by the sender and the receiver of a message; but we expect any central coordinator to be more complex, and thus less efficient than a local one.)

On Congestion and Scalability. The values cited above for t_{eval} and t_{eval}^C are for uncongested controllers. Obviously, when a controller is congested, its effective evaluation time, i.e., the latency for an event to be evaluated, can grow boundlessly. We examine here is the vulnerability to congestion of LGI and of CCM. We start with CCM.

With $t_{eval}^C = 2ms$, the central coordinator C will be congested when the frequency of messages exchanged between members of the group is larger than 500 per second. Such congestion would effect the latency of *every message* within the group, as all of them are mediated by C.

To understand the vulnerability of controllers under LGI to congestion, let us first make the following assumptions: (a) there are n members in the group, each with its own controller; and (b) the exchange of messages is uniform, so that each controller has to handle the same number of messages (in unit of time). Since each message generates a pair of regulated events under LGI, and given $t_{eval} = 1ms$, it follows that the controllers will be congested only when the frequency of messages exchanged between the members of the group is larger that $500n$. That is, the volume of message that can be handled by LGI, without congestion, is larger by a factor of n than under CCM.

This is one sense in which LGI is more scalable then CCM. Another sense emerges when we drop the uniformity assumption above. If an agent x sends or receives a lot of messages then its own controller will be congested if the frequency of these messages is larger than 1000 messages/second. But this congestion will effect *only* x *itself* and possibly some of its interlocutors, *but nobody else.* Under CCM, on the other hand, everybody is affected by the congestion of C, so that one overactive agent is able to slow down the entire group.

The relative overhead of centralized coordination. Using the typical values cited above, let us evaluate the relative overhead of conventional CCM, when there is no congestion: First, for communication across WAN, we plug t_{WAN} into Equation 3, which yields:

$$ro_{x,y}^{CC} = (t_{WAN} + t_{eval}^C)/t_{WAN} \approx 1 \tag{4}$$

The result of 1 for ro in this case is a good approximation as long as the controller C is not congested, and its evaluation time is around 2 ms, which is much smaller than t_{WAN}. The corresponding result for centralized coordination within a LAN, obtained by replacing our value for t_{WAN} with that for t_{LAN}, is 1.4.

These two values for the ro of uncongested centralized coordination—1.0 for WAN, and 1.4 for LAN—can be used as benchmarks, with which the relative overhead of LGI can be compared.

5.2 The Relative Overhead of LGI, Under Various Conditions

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. This provides us with flexibilities, which can often be used to minimize the overhead of LGI under various conditions. Also, the structure of the law itself can sometimes be used to reduce the overhead of LGI, by bypassing one of the controllers. We will consider here in detail the effect of these factors on the relative overhead of LGI across WAN, and we will also mention the corresponding results for LAN communication.

5.2.1 *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 then TCP communication. Applying Equations 2 and 1 to this situation and using our typical values, yields the following result for relative overhead:

$$ro_{x,y} = (2 * t_{eval} + 2 * t_{pipe})/t_{WAN} \approx .02 \qquad (5)$$

The corresponding result for relative overhead of LGI within a LAN would be 0.4. This is not quite as negligible as in the WAN case, but still affordable, and substantially smaller than the ro of centralized coordination within LAN, which we have already calculated as 1.4.
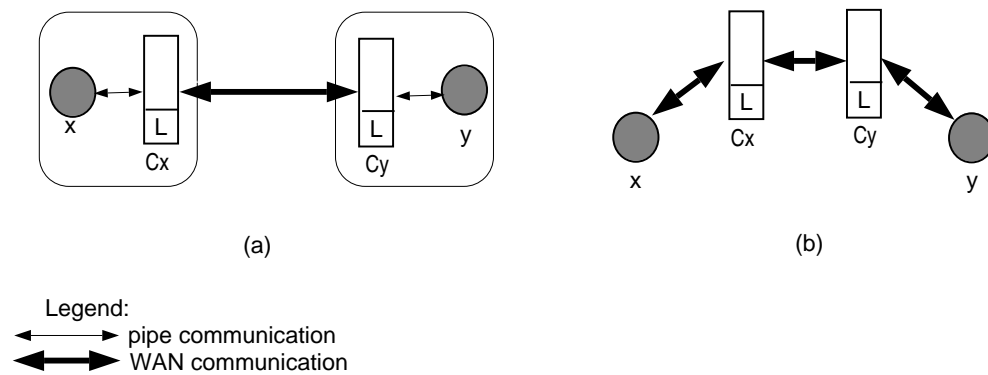

Fig. 6.   Placement of controllers:(a) on the same machine as the agent: (b) remote (across a WAN).

5.2.2 *Using Remote Controllers.* Local controllers should not be used when one is concerned about a controller being tempered with by its client. To enhance the security of LGI communication it would be appropriate to use controllers maintained by some kind of trusted authority, as explained in [Minsky and Ungureanu 1998a]. Such controllers would generally not reside in the LAN of their clients, or of each other, but might be anywhere in the Internet, as illustrated in part (b) of Figure 6.

To compute the relative overhead of such communication, we plug t_{WAN} for every communication time in Equation 1. This yields the following result for the relative overhead in this case:

$$ro_{x,y} = (2 * t_{WAN} + 2 * t_{eval})/t_{WAN} \approx 2 \qquad (6)$$

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

This is twice as large as the relative overhead of uncongested central coordinator (see Equation 4). However, as we shall see in the following section, even when security is an issue it is often possible, to reduce this overhead by exploiting the ability of several agents to share a single controller.

5.2.3 *Sharing Controllers.* Suppose that a single controller c is assigned to both x and y. The processing, with such a controller, of a regulated message from x to y is illustrated in part (a) of Figure 7. 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*-event[3]. As such, this placement scheme requires only one more TCP/IP message than required by unregulated message passing. Our formula for relative overhead would now yield

$$ro_{x,y} = (t_{WAN} + 2 * t_{eval})/t_{WAN} \approx 1 \qquad (7)$$

just as under centralized coordination.

Of course, this would not help with the communication of agent x with some other agent z, unless z is also assigned to controller c. One can, of course, emulate centralized coordination by assigning one controller c to all members of an $\mathcal{L}$-group. This will have the same overhead as a central coordinator, but will be equally unscalable.

Alternatively, one can adopt a *semi-centralized* controller-assignment strategy, assigning a single controller to a moderately sized cluster of agents that interact mainly between themselves. The relative overhead would then be 1 for inter-cluster communication, and 2 for intra-cluster communication—and the whole arrangement would be moderately scalable. In principle, such assignment of controllers to agents can be adjusted dynamically, adapting it to current communication patterns. However, the present implementation of Moses provides for only static assignment of controllers, so we did not experiment with dynamic adjustments.

---

[3]Controller-sharing works as follows: each controller maintains a table with 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.
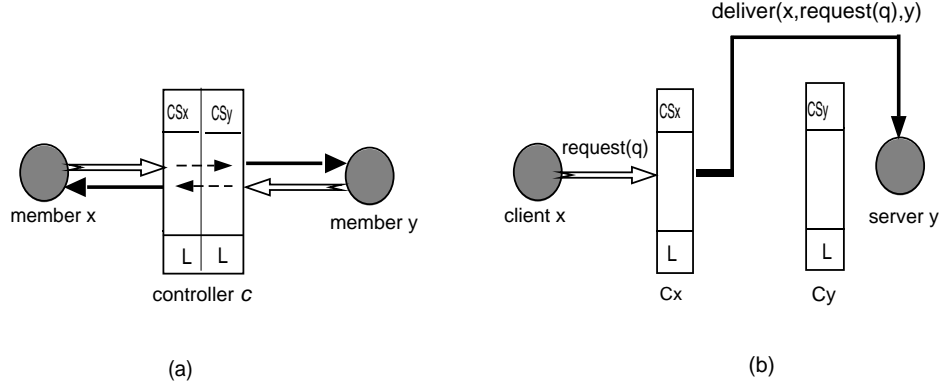
Fig. 7.   Using a single controller: (a) Agents x and y share the same controller c; (b) Bypassing a controller: message `request(q)` is directly delivered to the database server.

5.2.4 *Bypassing a Controller.* So far we had each message from x to y generate two controlled events, *sent* and *arrived*, which were evaluated by the controller assigned to x and to y, respectively. We have just seen, that if we assign one controller to both agents, we save a controller-controller communication, but still have two law- evaluations to perform. Here we will see that in some cases it is possible to bypass a controller altogether, resulting in a bigger saving still.

This is possible when dealing with a law $\mathcal{L}$ that prescribes that when a message m sent by x arrives at y, it is to be delivered to y without having any other effect, and independently of the control-state of y. With such a law, the controller of y can be *bypassed* by having the controller of x deliver message m directly to y itself, thus reducing the overhead by one message passing and one law-evaluation.

An example of this situation is provided by our $\mathcal{BC}$ law, which prescribes that any message arriving at a database server is to be delivered without further ado. The server's controller can be bypassed in this case simply by replacing the pair of rules $\mathcal{R}4$ and $\mathcal{R}5$ in Figure 2 with the following rule:

```
R4″.  sent(X,request(Query),D) :- budget(B)@CS, B > 0,
         do(dcr(budget(B),1)),
         do(deliver(X,request(Query),D)).
```

This rule calls for the request to be delivered directly to the database server d by the client's controller $\mathcal{C}_x$, bypassing the server's controller, as shown in part (b) of Figure 6. This reduces the overhead by one message and one law evaluation. Assuming that the controller of x is local we get the following result for relative overhead for WAN communication.

$$ro_{x,y} = (t_{WAN} + t_{eval})/t_{WAN} \approx .01 \qquad (8)$$

The corresponding result for LAN communication is 0.2.

Finally, we note that in many cases it should be possible to detect automatically, from the law in question, that the destination-controller can be bypassed. It would then not be necessary to change the law manually to perform a remote delivery,

as we have done above. But the present version of Moses does not perform such optimization automatically.

5.3 On the Performance of the Current Moses Toolkit

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—the present t_{eval} is approximately 3.5 ms. As already pointed out, we expect future versions of the controller to be much faster, in the following ways: First, by compiling the controller, rather than interpreting it. The development of good Java compilers is underway in many places, and is expected to achieve near native performance [Halfhill 1998]. This factor alone should reduce t_{eval} by a factor of 3 or 4 [Fitzgerald et al. 1999].

Second, the performance of our controllers should be improved significantly with the advent of better implementation of Java core libraries. In particular, it has been recently shown in [Heydon and Najork 1999] that excessive synchronization performed by low level classes can lead to a substantial performance degradation. For example, some of the I/O classes are synchronized. Since each such call requires a lock acquisition, the controller is spending much of its time acquiring locks for objects that are (usually) used by only one thread. This situation can be easily solved by providing both synchronized and unsynchronized versions for different classes.) Another cause of performance degradation is excessive heap allocation [Klemm 1999; Halfhill 1998] which in turn leads to more time spent doing garbage collection.

Finally, we expect to replace Prolog as a language for writing laws with a simpler language to interpret. All told, we believe that our expectation for t_{eval} to become 1ms is quite conservative.

Experimental Results. 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. But they generally conform to the predictions in Section 5.2, when the value of 3.5ms is used for t_{eval}. We report here in detail only our measurement of the performance of the controller itself.

We have carried out a series of experiments aimed to measure the controller performance *per se*, 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. Specifically, we used events of the form `sent(`c_i`,request(query),`s_j`)` where c_i and s_j are members in $\mathcal{BC}$-group. 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 (each c_i appears as sender in n_e/n `sent` events).
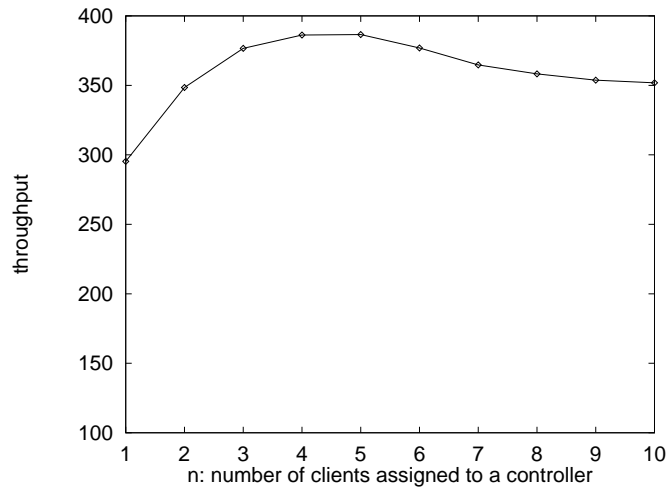
Fig. 8.   The performance of a controller under a heavy load

Figure 8 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 with the number of clients which varies between 290 events/s and 390 events/s. 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. The experiment shows, though, that the controller performs well even for large number of agents: for example, if there are 10 clients assigned to the controller the deprecation in performance is less than 10%.

6. RELATED WORK

There has been a growing interest in coordination in recent years, and a variety of different, and quite powerful, coordination mechanisms have been devised. We provide here a short overview of these mechanisms, and we conclude by placing this paper in the context of previous work by the authors.

To the best of our knowledge, none of the coordination mechanisms proposed so far, satisfies our three principles for coordination in open groups. This is not surprising, as most coordination mechanisms have been devised for *close-knit groups*, generally written in a single language, and spawned by a single program, which do not require all these three principles. This is certainly true for Linda [Carriero and Gelernter 1992], one of the first, and most influential, explicit attempts at coordination. Linda features very powerful and elegant coordination primitives, but *it provides for no explicit statement of a policy*. So, if a group of agents are to coordinate effectively via a tuple-space (a basic Linda concept) they all need to *internalize* some kind of common policy. (Many Linda-based mechanisms, like Sonia [Banville 1996], for example, share this property of Linda.) Of course, no

such internalization can be relied on in an open group—which is the reason for our first and third principles.

There is a collection of mechanisms that do provide for an explicit coordination policy, but only for groups of agents spawned by a single program or, at least, written in a specific language. These techniques are not applicable to open groups, in our sense of this term, where the group members may have been written independently, possibly in different languages. Some of these mechanisms, including Actors [Frolund and Agha 1993; Agha 1997], Contracts [Holland 1993], and Composition Filters [Aksit et al. 1993], do not depend on centralized control and are thus, potentially scalable. Others, like Gamma and the Chemical Reaction Model [Banatre and Le Metayer 1996], LO [Andreoli 1996], and COOLL [Castellani and Ciancarini 1996], do not concern themselves with scalability, and rely on centralized coordinators or on broadcasting.

But even the few mechanisms that have been specifically designed for open systems, did not adopt all our principles, possible because they used a different definition of openness. We will cite here two such coordination mechanisms . First, Objective Linda (OL) [Kielmann 1996] attempted to adapt Linda for open systems, mostly by introducing a hierarchy of object spaces, which provides a degree of access control. However, like Linda, OL does not provide for the specification of an explicit policy that govern the interaction of agents with the tuple spaces. This is clearly unsuitable for an heterogeneous group, whose members are not familiar with each other, and cannot be trusted to internalize a common set of rules. Another coordination mechanism intended for open groups is Coordination Language Facility (CLF) [Andreoli et al. 1998]. Coordination is carried out in CLF by means of a central coordinator. This coordinator performs sophisticated atomic negotiation protocol, by sending out instruction to the group members (called "participants"), so that part of the coordination is carried out in a non-centralized manner. But CLF provides no assurances that the participants actually carry out the instructions of the coordinator correctly. Thus, CLF lacks the local enforcement provided by LGI.

The present concept of LGI is based squarely on the concept of Law Governed Architecture for Distributed System (LGAD) introduced by one of the authors [Minsky 1991]. But there are several significant differences between LGI and its precursor, which make LGI more general and far more practical. First, LGAD has been an attempt to define the architecture of an entire distributed system as an enforced law. This overarching concept of law turned out to be quite impractical for modern systems operating over the Internet, and have never been fully implemented. It has been replaced in LGI with a law of group within a system, whose membership is voluntary and may change dynamically. This modification made LGI implementable, it changed the nature of coordination and control under LGI in a fundamental way, and it made LGI useful for a wide range of applications (which have been explored in a series of conference papers (see Section 4)).

Second, the implementation of LGI allowed us to address in this paper issues that could not have been addressed in [Minsky 1991]. These include deployment techniques, the analysis of the overhead of LGI (discussed in Section 5), and optimization techniques, such as the sharing of controllers, the placement of controllers, and remote delivery.

Finally, the law under LGI is considerably more expressive due to its concepts of *exception*, which helps in making policies more robust and fault tolerant; and the concept of *enforced obligation*, which allows one to ensure liveness properties. Our concept of enforced obligations is based on a concept of obligation introduced in [Minsky and Lockman 1985], which was too general for practical implementation. The adaptation of that concept to LGI was done in collaboration with Dr. Jerry Leichter. Our notion of obligation has been used later in by Roscheisen in his thesis [Roscheisen and Winograd 1996]. (For related work on obligation in the context of deontic logic, see Section 3.1).

7. CONCLUSION

We have argued that the coordination within open groups of autonomous and distributed agents, and the control of such groups, call for the following principles to be satisfied: (1) coordination policies need to be enforced; (2) the enforcement needs to be decentralized; and (3) coordination policies need to be formulated explicitly—rather than being implicit in the code of the agents involved—and they should be enforced by means of a generic, broad spectrum mechanism; and (4) it should be possible to deploy and enforce a policy incrementally, without exacting any cost from agents and activities not subject to it.

We presented a mechanism called law-governed interaction (LGI), currently implemented by the Moses toolkit, which has been designed to satisfy these principles. We have shown that LGI is at least as general as a conventional centralized coordination mechanism (CCM), and that it is more scalable, and generally more efficient than CCM. But our presentation of the current LGI model is not quite complete. In particular, we did not describe here the details of implicit groups, briefly mentioned in Section 2.1, and we did not discuss at all the very important issue of interoperability between different $\mathcal{L}$-groups. These issues, and some others, will be discussed in a subsequent paper.

REFERENCES

AGHA, G. 1997. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. NAJM AND J.-B. STEFANI Eds., *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions. Chapman and Hall.

AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. 1993. Abstracting object interactions using composition filters. In *ECOOP'93, LNCS 791* (1993), pp. 152–184.

ANDREOLI, J.-M. 1996. Coordination in LO. In J.-M. ANDREOLI, C. HANKIN, AND D. LE METAYER Eds., *Coordination Programming*, pp. 42–64. Imperial College Press.

ANDREOLI, J.-M., PACULL, F., PAGANI, D., AND PARESCHI, R. 1998. XPECT: Multiparty negotiation of dynamic distributed object services. *Journal of the Science of Computer Programming*, 179–203.

BANATRE, J.-P. AND LE METAYER, D. 1996. Gamma and the chemical reaction model: Ten years after. In J.-M. ANDREOLI, C. HANKIN, AND D. LE METAYER Eds., *Coordination Programming*, pp. 3–41. Imperial College Press.

BANVILLE, M. 1996. Sonia: an adaptation of Linda for coordination of activities in organizations. In P. CIANCARINI AND C. HANKIN Eds., *Coordination Languages and Models*, Lecture Notes in Computer Science, pp. 57–74. Springer-Verlag. Number 1061.

BREWER, D. AND NASH, M. 1989. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium in Security and Privacy* (1989). IEEE Computer Society.

BROWN, M. 1998. 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).

CARRIERO, N. AND GELERNTER, D. 1992. Coordination languages and their significance. *Communications of the ACM 35*, 2 (Feb.), 97–107.

CASTELLANI, S. AND CIANCARINI, P. 1996. Enhancing coordination and modularity mechanisms for a language with objects-as-multisets. In P. CIANCARINI AND C. HANKIN Eds., *Coordination Languages and Models*, Lecture Notes in Computer Science, pp. 89–106. Springer-Verlag. Number 1061.

CLOCKSIN, W. AND MELLISH, C. 1981. *Programming in Prolog*. Springer-Verlag.

FITZGERALD, D., KNOBLOCK, T., RUF, E., STEENSGAARD, B., AND TARDITI, D. 1999. Marmot: an optimizing compiler for Java. Technical report (June), Microsoft Research.

FROLUND, S. AND AGHA, G. 1993. A language framework for multi-object coordination. In *ECOOP'93 Conference Proceedings, LNCS 707* (1993), pp. 346–360. Springer-Verlag.

GAL, A., MINSKY, N., AND UNGUREANU, V. 1999. Regulating agent involvement in inter-enterprise electronic commerce. In *The 19th International Conference on Distributed Computing Systems (ICDCS)* (June 1999).

HALFHILL, T. 1998. How to soup up Java. *Byte*. website: http://www.byte.com/art/9805/sec5/art1.htm.

HEYDON, A. AND NAJORK, M. 1999. Performance limitations of the Java core libraries. In *Proceedings of the ACM 1999 Java Grande Conference* (June 1999).

HOLLAND, I. 1993. Specifying reusable components using contracts. In *ECOOP'92*, Lecture Notes in Computer Science, pp. 287–308. Springer-Verlag. Number 615.

KENT, S., MAIBAUM, T., AND QUIRK, W. 1993. Formally specifying temporal constraints and error recovery. In *Proceedings of the IEEE Int. Symp. on Requirement Engineering* (San Diego, CA, Jan. 1993), pp. 208–215.

KIELMANN, T. 1996. Designing a coordination model for open systems. In P. CIANCARINI AND C. HANKIN Eds., *Coordination Languages and Models*, Lecture Notes in Computer Science, pp. 267–284. Springer-Verlag. Number 1061.

KLEMM, R. 1999. Practical guideline for boosting Java server performance. In *Proceedings of the ACM 1999 Java Grande Conference* (June 1999).

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

MALONE, T. AND CROWSTON, K. 1994. The interdisciplinary study of coordination. *ACM Computing Surveys 26*, 1 (March), 87–119.

MALONE, T., CROWSTON, K., J., L., AND PENTLAND, B. 1993. Tools for inventing organizations: Toward a handbook of organizational processes. *Proceedings of Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 72–82.

MEYER, J. J. C., WIERINGA, R. J., AND F.P.M., D. 1998. The role of deontic logic in the specification of information systems. In J. CHOMICKI AND G. SAAKE Eds., *Logic for Databases and Information Systems*. Kluwer.

MINSKY, N. 1991. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*.

MINSKY, N. AND LOCKMAN, A. 1985. Ensuring integrity by adding obligations to privileges. In *Proceedings of the 8th International Conference on Software Engineering* (Aug. 1985), pp. 92–102.

MINSKY, N., MINSKY, Y., AND UNGUREANU, V. 2000a. Access control for Linda-like tuple spaces. *Journal of Applied Artificial Intelligence*. (to be published).

MINSKY, N., MINSKY, Y., AND UNGUREANU, V. 2000b. Making tuple spaces safe for heterogeneous distributed systems. In *2000 ACM Symposium on Applied Computing— the Coordination Track, Como, Italy* (March 2000). (to be published—available through `http://www.cs.rutgers.edu/~minsky/`).

MINSKY, N. AND UNGUREANU, V. 1997. Regulated coordination in open distributed systems. In D. GARLAN AND D. L. METAYER Eds., *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282* (September 1997), pp. 81–98.

MINSKY, N. AND UNGUREANU, V. 1998a. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)* (May 1998), pp. 322–331.

MINSKY, N. AND UNGUREANU, V. 1998b. Unified support for heterogeneous security policies in distributed systems. In *7th USENIX Security Symposium* (January 1998).

MINSKY, N., UNGUREANU, V., WANG, W., AND ZHANG, J. 1996. Building reconfiguration primitives into the law of a system. In *Proc. of the Third International Conference on Configurable Distributed Systems (ICCDS'96)* (March 1996). (available through `http://www.cs.rutgers.edu/~minsky/`).

NEEDHAM, R. 1994. Denial of service: An example. *Communications of the ACM*, 42–46.

RIVEST, R. 1992. The MD5 message digest algorithm. Technical report (April), MIT. RFC 1320.

ROSCHEISEN, M. AND WINOGRAD, T. 1996. A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 1996).

S., F. M. 1993. An implementation of bounded obligations. In *Proceedings of the 8th Knowledge Based Software Engineering Conference* (Chicago, Ill, Sept. 1993), pp. 114–122.

SCHNEIER, B. 1996. *Applied Cryptography*. John Wiley and Sons.