Law-Governed Internet Communities

Xuhui Ao, Naftaly Minsky, Thu D. Nguyen, and Victoria Ungureanu

Rutgers University, New Brunswick, NJ 08903, USA,
{ao,minsky,tdnguyen,ungurean}@cs.rutgers.edu

Abstract. We consider the problem of coordination and control of large heterogeneous groups of agents distributed over the Internet in the context of Law-Governed Interaction (LGI) [2, 5]. LGI is a mode of interaction that allows a group of distributed heterogeneous agents to interact with each other with confidence that an explicitly specified policy, called the *law* of the group, is complied with by everyone in the group.
The original LGI model [5] supported only *explicit* groups, whose membership is maintained and controlled by a central server. Such a central server is necessary for applications that require each member of the group to know about the membership of the entire group. However, in the case where members do not need to know the membership of the entire group, such a central server can become an unnecessary performance bottleneck, as group size increases, as well as a single point of failure.
In this paper, we present an extension to LGI allowing it to support *implicit groups*, also called *communities*, which require no central control of any kind, and whose membership does not have to be regulated, and might not be completely known to anybody.

1 Introduction

We consider the problem of coordination and control for large heterogeneous groups of agents distributed over the Internet in the context of Law-Governed Interaction (LGI) [2, 5]. LGI is a mode of interaction that allows a group of distributed heterogeneous agents to interact with each other with confidence that an explicitly specified policy, called the *law* of the group, is complied with by everyone in the group. LGI has been designed specifically to satisfy the following principles, which we consider critical for coordination in large heterogeneous systems: (1) coordination policies need to be formulated explicitly rather than being implicit in the code of the agents involved, (2) coordination policies need to be enforced, and (3) the enforcement needs to be decentralized, for scalability. LGI has been implemented in a toolkit called Moses, which has been applied to a broad range of coordination and control applications, including: on-line reconfiguration of distributed systems [6], security [4], and electronic commerce [3].

A group of agents interacting via LGI under a given law $\mathcal{L}$ is called an $\mathcal{L}$-group. LGI distinguishes between two kinds of $\mathcal{L}$-groups, called *explicit* and *implicit* groups, that differ in the manner in which a group is deployed and in the management of its membership. Explicit groups have been discussed in detail in [5]. The purpose of this paper is to introduce implicit groups, also called *communities*, which are more general than explicit ones, and more suitable for very large groups of heterogeneous agents operating over the Internet.

Currently, an explicit $\mathcal{L}$-group $\mathcal{G}$ is established in Moses by creating a distinguished agent called the *secretary* of $\mathcal{G}$, denoted as $\mathcal{S}_{\mathcal{G}}$, and defining into it the law $\mathcal{L}$, and specifying the initial membership and structure of $\mathcal{G}$. Subsequent to its initialization, $\mathcal{S}_{\mathcal{G}}$ serves as a "gateway" to the group by admitting new members into it, subject to law $\mathcal{L}$. $\mathcal{S}_{\mathcal{G}}$ also functions as a name-server for the group, helping members to find each other's location, and to verify mutual memberships in the same group.

Such a secretary is necessary whenever the entire membership of the group needs to be known, and it is appropriate for relatively small groups. This is the case, for example, for a group operating under a token-ring protocol, where the structure of the group, i.e., the placement of its members along a ring and the existence of a single token among the members of the ring, are essential to protocol. This ring structure can be defined by the secretary of the group as its initial state, and, as demonstrated in [6], can be maintained as an invariant, even if the membership of the group is allowed to change dynamically.

But such group management is neither necessary nor appropriate where no knowledge of the entire group membership is required, or available, and where the size of the group is too large to be comfortably handled by a single secretary. An everyday example for such a situation is provided by the group of all car drivers in a given city. All these drivers must obey the traffic laws of the city, but generally there is nobody that knows the names of all these drivers, or their total membership. Such conditions are becoming increasingly common in modern distributed computing, as is illustrated by the following example.

Consider a distributed set of databases servers that provides access to an heterogeneous set of clients. Suppose that for a client to consult an item in a database or to update it, it needs to lock the item first. It is possible for a single agent to maintain locks for several items (at several databases) at a time. It is well known that this activity would be *serializable* if the following kind of two-phase locking (TPL) protocol is strictly observed by all clients [10]:

> New locks cannot be acquired after the first release of a lock (until the agent has released all locks that it currently holds). That is, each transaction (representing some set of changes) is divided into two phases: a *growing phase* of locking, and a *shrinking phase* of releasing locks. A locked resource can be used during both phases.

While this protocol can be enforced by a *central coordinator* that mediates the interaction between the distributed set of servers and their clients, such coordination would be quite unscalable. Under LGI, on the other hand, this protocol can be formulated as a law $\mathcal{TPL}$ that is enforced locally at each client, allowing for scalability, provided that the set of servers and their clients is not maintained as an *explicit* $\mathcal{TPL}$-group. Because the number of clients in this case might be very large, the requirement that each of them enters the group via a single secretary would create a bottleneck and a dangerous single point of failure. Moreover, the maintenance of an explicit group seems quite unnecessary here: none of the agents in this case needs to know the membership of the entire $\mathcal{TPL}$-group. In fact, the clients can be quite oblivious to the very existence of other clients.

Our concept of implicit groups, or communities, has been designed to deal with this kind of siltation. Broadly speaking, a community operating under law $\mathcal{L}$—or, an $\mathcal{L}$-community—is defined as the set of all agents that happen to be operating under law $\mathcal{L}$. Such a community is never formally established and there is no formal admission into it. Anybody can become a member of this community, simply by adopting the law $\mathcal{L}$ when using LGI.

While agents operating in an implicit group may not need to know the membership of the entire group, they still need to interact with one another—this, after all, is the purpose of a community. Thus, to support implicit groups, we need to provide the following capabilities (which for an explicit groups are provided by its secretary): (a) means for an agent operating under law $\mathcal{L}$ to ensure that its interlocutors are also operating under $\mathcal{L}$, i.e., that they indeed belong to the same community, (b) a convenient naming scheme that assigns a unique name to each member of an $\mathcal{L}$-community, and (c) means for supplying certain agents in an $\mathcal{L}$-community with *exclusive* privileges. The importance of these capabilities, and the manner in which we have extended LGI to provide for them, is described in this paper.

The remainder of the paper is organized as follows. In Section 2, we give a brief description of LGI and discuss how agents can join an $\mathcal{L}$-community and how they can name and locate each other. In Section 3, we describe how certificates can be used to supply certain agents in an $\mathcal{L}$-community with exclusive privileges. We conclude the paper in Section 4.

2 Law-Governed Community

2.1 Law Governed Interaction – An Overview

LGI is a mode of interaction that allows a group of distributed heterogeneous agents to interact with each other with confidence that an explicitly specified policy, called the *law* of the group, is complied with by everyone in the group. We call such a group of agents a community, or more specifically, an $\mathcal{L}$-community, where $\mathcal{L}$ is the law of the community. LGI does not assume any knowledge about the structure and behavior of the members of a given $\mathcal{L}$-community: LGI only deals with the interaction between these agents. However, LGI does maintain some state for each member of the community, which is called the *control-state*. Such per-agent states enable the law to differentiate between specific agents, and to be sensitive to changes in their states, which are, themselves, subject to the law. The control-state, whose semantics for a given community is defined by its law, could represent such things as the role of an agent, various kinds of privileges and tokens it carries, and dynamic identification of the state of computations in which the agent is involved.

The Concept of Law: The *law* $\mathcal{L}$ of a community is an *explicit and enforced* set of "rules of engagement" between members of this community, regulating the interaction between them via what we call $\mathcal{L}$-messages. More specifically, the law of a community $\mathcal{C}$ regulates certain types of events occurring at members of $\mathcal{C}$, mandating the effect that any such event should have; this mandate is called the *ruling* of the law for a given event. In LGI, events subjected to regulation include message *sends* and *receipts* of $\mathcal{L}$-messages, the *coming due of an obligation* previously imposed on a given object. Two additional types of events regulated under LGI will be introduced in this paper.

The ruling of the law for a given event can involve the execution of *operations*, called *primitive operations*. LGI currently support primitive operations for testing the control-state of an agent and for its update, operations on messages, and some others – a sample of primitive operations (written in Prolog) is presented in Figure 1.

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

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

Fig. 1. Some primitive operations in LGI

Generally speaking then, a law $\mathcal{L}$ regulates the exchange of $\mathcal{L}$-messages between members of an $\mathcal{L}$-community based on the control-states of the participants. Furthermore, it can mandate side effects for message-exchanges such as the modification of the control-states of the sender and/or receiver of a message, and the emission of extra messages (for monitoring purposes, say).

On The Local Enforceability of Laws: Although the law $\mathcal{L}$ of a community $\mathcal{C}$ is *global* in that it governs the interaction between all members of $\mathcal{C}$, it is enforceable *locally* at each member of $\mathcal{C}$ because:

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

The fact that a law is enforced at all agents of a community gives LGI its necessary global scope, establishing a *common* set of ground rules for all members of $\mathcal{C}$ and providing them with the ability to trust each other, in spite of the heterogeneity of the community. The locality of law enforcement, however, critically enables LGI to scale with community size.

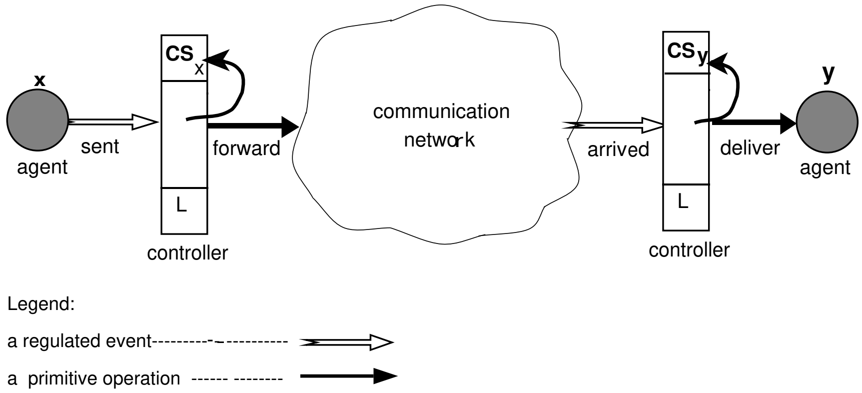

Fig. 2. Law enforcement in Moses.

On the Current Implementation of LGI, via Moses: We have implemented LGI via the *Moses* toolkit. In Moses, laws are written as Prolog programs and the law $\mathcal{L}$ of an $\mathcal{L}$-community $\mathcal{C}$ is enforced by a set of trusted agents called *controllers*, which mediate the exchange of $\mathcal{L}$-messages between members of $\mathcal{C}$. Each member `x` of $\mathcal{C}$ must request some controller to maintain its control-state $\mathcal{C}_x$ and enforce $\mathcal{L}$ on its behalf. Figure 2 illustrate law enforcement in Moses, where controllers are logically placed between the members of $\mathcal{C}$ and the communication medium.

A controller is *generic* in that it can interpret and enforce any well formed law. In the current implementation, a controller operates as an independent process and may be placed on the same machine as its clients or on some remote machine. To help agents locate and contact controllers, we have implemented a *controller-naming* server, which maintains a set of available controllers. To be effective, for a widely distributed environment (such as the Internet), this set of controllers need to be well dispersed geographically, so that it would be possible to find a controller reasonably close to any prospective client.

2.2 Engaging in an $\mathcal{L}$-Community

For an agent x to be able to exchange $\mathcal{L}$-messages with other members of an $\mathcal{L}$-community, it must: (a) find an LGI controller, and (b) notify this controller that it wants to use it, under law $\mathcal{L}$. We will discuss these two steps below, and then explain how and why x can trust its interlocutors to observe the same law $\mathcal{L}$.

Locating an LGI Controller: As already discussed, the Moses toolkit includes a controller-naming server, which can be used to maintain a set of active controllers. This server provides the address (host and port) of the available controllers to any agent that wishes to engage in LGI. One may have any number of such servers so that controllers can be distributed in different regions of the Internet. Efficiency-wise, x would do best by selecting a controller closest to it (to minimize the overhead of forwarding $\mathcal{L}$-messages through the controller). But functionally, one is free to choose a controller anywhere on the Internet, and several agents may share a single controller, without knowing of each other.

Adopting a law: Upon selecting a controller C, x would send C the message

```
adopt(law,name),
```

where `law` is the law that it wants to adopt, and `name` is the name that it wants to be known by. The argument `law` can take the form of either the text of the law to be adopted or the name of such a law, given to it by a specified *law-repository* service, which is another tool provided by Moses—we will not discuss here the details of this service but rather assume that the text of the entire law is always passed to the controller.

When controller C receives the `adopt` message, it checks the supplied law for syntactic validity, and the chosen name for uniqueness among the names of all current agents handled by C. If these two conditions are satisfied, and if C is not already loaded to capacity, it will set up a staring control-state for agent x, as specified in the preamble of the law adopted by x, allowing x to start operating under this law[1].

The basis for trust between members of a community: The point of adopting a given law is to be able to interact with other agents operating under it. For this, one needs to be able to locate such agents—which will be discussed in the next section – and one needs to be able to trust its interlocutors to operate under the same law. More specifically, 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 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, $\mathcal{L}$-messages cannot be forged.

To ensure that a message forwarded by a controller $\mathcal{C}_x$ under law $\mathcal{L}$ would be handled by another controller $\mathcal{C}_y$ operating 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 that transform any string into a considerably smaller bits sequence with high probability that two strings will not collide [7, 9].) $\mathcal{C}_y$ would accept this as a valid $\mathcal{L}$-message under $\mathcal{L}$ if and only if `H` is identical to the hash of its own law.

[1] If any one of these conditions is not satisfied, then `x` would receive an appropriate diagnostic, and will be able to try again.

With respect to the correctness of the controllers, if an agent is not concerned with malicious violations, then it can trust a controller provided by our controller-naming service, or a controller provided by the operating system—just like we often trust various standard services on the Internet, such as DNS and gateways. When malicious violations are a concern, however, the validity of controllers and of the host on which they operate needs to be certified. In this case, the controller-naming service needs to operate as a *certifying authority* for controllers. Furthermore, 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 ([3]).

2.3 Naming Within A Community

As already mentioned, when an agent joins a community, it must have a way of *naming* and *locating* other members of the community. After all, one joins a community only if one wishes to interact with some of its members. In the case of an explicit group, where there is a server, called secretary, that maintains group membership, then naming is easy. The secretary simply acts as a naming and locating service, negotiating with agents wishing to join the group in order for each agent to have a unique name within that group. In the case of a community, however, there is no such server. Thus, we need to develop a naming scheme to support communities.

In a distributed environment like the Internet, any naming scheme must meet the following requirements: (1) it must be possible to locally choose names that are globally unique, and (2) given a name, it must be possible (and easy) to locate the controller of the named agent

In addition, it would be convenient if names are human readable, selectable and well organized (so that we can easily understand names and not require translations from the machine representation to a human-parsable representation).

To satisfy the above requirements, we use a naming scheme that is very similar to current e-mail addressing. In our scheme,

```
memberName = localName@domainName
```

where `domainName` is the Internet host name of the controller of the agent and `localName` is the name negotiated between the agent and its controller when the agent first adopted the law of the community at its controller. Since `localName` is unique to the controller and the controller's host name is unique in the Internet, we can be sure that the whole name is globally unique.

Note that this simple naming scheme satisfies all of our requirements. Choosing a name is entirely a local operation between an agent and its controller. Yet, because we are leveraging the globally unique host names of controllers, a name that is chosen locally is guaranteed to be unique globally. Given an agent's name such as `bill@athos.rutgers.edu`, we simply leverage the already existing DNS service to locate the server `athos.rutgers.edu`. Finally, these names are certainly human readable—in fact, they are very familiar because current users already use this naming scheme every day for e-mail.

Since the member name of a community is globally unique, one can have a bunch of community name servers all over the Internet to publish the member names of the community. So one agent can easily locate other members in that community.

2.4 The $\mathcal{TPL}$ Law—an Example

To illustrate the nature of $\mathcal{L}$-communities , and the structure of their laws, we now show how the two phase locking (TPL) policy introduced informally in Section 1 can be formalized into a law, called $\mathcal{TPL}$ and specified in Figure 3. The rules of this law are followed by comments (in italic), which, together with the following discussion, should provide the reader with some understanding of the nature of LGI laws. Besides its rules, every law under LGI has a *preamble* that contains an `initialCS([...])` clause, specifying the initial control state of every agent that adopts this law. The preamble may also contain other types of clauses, one of which will be discussed in the following section.

Under this particular law, all new agents start with an empty control-state, and are, thus, indistinguishable from each other. But any agent can designate itself as a *server*, simply by sending a message `role(server)` to itself; which, by Rule $\mathcal{R}1$, would cause the term `server` to be added to its control-state. (In the following section, we will present constraints on such self appointments.) We also note here that under this law, a term `shrinking` is added to the control state of an agent `x` the first time it issues a request to update a resource, to record that it entered the second phase of a transaction (the *growing* phase is designated by the absence of the `shrinking` term in the control state). We will now discuss how resources can be locked under this law, how they can be used, and finally, how they are unlocked.

First, by Rule $\mathcal{R}2$, if a client `x` issues a request to `lock` a resource `r`, then this message is forwarded to the destination server `s` only if `x` has not yet entered in the shrinking phase. Also a term `lock(r,s,pending)` is added to the control state of `x`, to record that `x` has issued a `lock` request for `r`. Now, if the resource is available, the server `s` is expected to respond with a `locked(r)` message, which it can do by Rule $\mathcal{R}7$. When `x` receives such a message, then, by Rule $\mathcal{R}3$, the term `lock(r,s,pending)` is replaced by a term `lock(r,s,granted)`.

Once one has the term `lock(r,s,granted)` in its control-state, indicating a lock over resource `r`, one can, by Rule $\mathcal{R}4$, send service requests to `s` for this resource. Note that the response of the server to such a request is left unregulated by this law.

A client `x` may release any resource `r` that it previously locked, by sending an `unlock(r)` message to the server `s` managing `r`. Such messages are regulated by Rule $\mathcal{R}5$ which mandates that the corresponding `lock` term is to be removed from the control state of `x`. Also, if this is done during the growing phase, i.e., when the `shrinking` term is absent form the control-state of `x`, then the `shrinking` term is added, indicating the beginning of the shrinking phase. Finally, if `r` was the last resource hold by `x` then, the term `shrinking` is removed from the control state, thus allowing `x` to start a new transaction, in its growing phase.

Note that Rule $\mathcal{R}6$ allow servers to receive arbitrary messages. Although servers do not provide here any specific control, the very fact that they use $\mathcal{TPL}$-messages to communicate with their clients, forces agents that want to be their clients to use the same law, which is what ensures that policy TPL is satisfied.

Discussion: We end this description of law $\mathcal{TPL}$ with two additional comments. First, the capacity of LGI to provide control at the *client side* is essential for the implementation of this protocol. Pushing enforcement on the *server side*, like conventional mechanisms do, would make the support of this protocol very difficult, if at all possible in the case considered here, where servers are distributed and may belong to possible different administrative domains. Such an implementation would require each server to know what resources, if any, the agent locked from other servers in the past, and even what it is requesting from them concurrently.

Second, the correctness of two phase locking protocol rests on the assumption that at any given time only one client may hold a lock on a resource. $\mathcal{TPL}$ law does not attempt to regulate lock management, thus implicitly trusting the servers to respond correctly to `lock` messages. Since there are no restrictions on becoming a server, such an assumption is appropriate, only when one trusts all agents to be non-malicious and bug-free. We will see in the next Section how this over-reliance on the correct behavior of agents can be removed.

3 Making Some Agents More Equal Than Others

An important (but not always desirable) property of communities under LGI as described so far is that they are intrinsically *egalitarian*. That is, it is not possible to endow certain agents of a community with *exclusive privileges*.

The need for such exclusive privileges is evident from our locking example. Law $\mathcal{TPL}$ allows anybody to become a server, simply by sending the `role(server)` message to itself—which everybody is allowed to do by Rule $\mathcal{R}1$, and which causes a `server` term to be added to the control-state of the sender. But since servers must be trusted to actually lock resources upon a valid request, it would be useful to be able to allow only certain trusted agents to play this role.

For further illustration of the need to provide some agents with more power than others, consider the following elaboration of our example. Suppose that for the sake of load balancing or security or both, we would like to introduce *brokers* into our $\mathcal{TPL}$-community to mediate between the clients and the servers. Under the revised community, to be governed by law $\mathcal{TPL}'$ (introduced below), a client would need a broker's referral to a server in order to use it; and each broker is to be responsible for a subset of servers. Under law $\mathcal{TPL}'$, a broker is to be designated as such by the term `broker` in its control-state, and we will see later

Preamble:
initialCS([]).
The initial control state of all members is empty.

$\mathcal{R}1$. `sent(X,role(server),_) :- do(+server).`
Under this law an agent can act a server—i.e., have a term `server` *in its initial control state—simply by sending a message* `role(server)`.

$\mathcal{R}2$.
```
sent(C,lock(R),S) :- !shrinking@CS,
        do(forward),
        do(+lock(R,S,pending)).
```
A message to lock resource `R` *is forwarded to its destination, only if the sender* `C` *is not in the shrinking phase. Also a term* `lock(R,S,pending)`*, denoting the pending status of this request, is added to the control state of* `C`.

$\mathcal{R}3$.
```
arrived(S,locked(R),C) :- do(lock(R,S,pending) ←lock(R,S,granted)),
        do(deliver).
```
If lock for resource `R` *is granted to an agent* `C` *then a term* `lock(R,S, pending)` *is replaced by* `lock(R,S,granted)` *to record that the lock has been acquired for* `R`*. The message then is delivered to the agent itself, in order to keep it informed.*

$\mathcal{R}4$.
```
sent(C, request(R,Param),S) :- lock(R,S,granted)@CS,
        do(forward).
```
A request by client `C` *regarding resource* `R` *is forwarded to server* `S` *only if the lock for* `R` *has been granted by* `S` *to* `C` *.*

$\mathcal{R}5$.
```
sent(C, unlock(R),S) :- lock(R,S,granted)@CS,
        do(-lock(R,S,_)),
        (!shrinking@CS →do(+shrinking);true),
        (!lock(_,_,_)@CS →do(-shrinking);true),
        do(forward).
```
An unlock request is forwarded if the sender currently hold this lock; also, the correspondent `lock` *term is removed from the control state of the issuer. If this agent is not yet in its shrinking phase, it enters this phase by adding the term* `shrinking` *to its CS; and if* `R` *is the last locked resource held by* `C`*, the term* `shrinking` *is removed.*

$\mathcal{R}6$.
```
arrived(C,M,S) :- server@CS,
        do(deliver).
```
Any message that arrives at a server is delivered, without further ado.

$\mathcal{R}7$.
```
sent(S,locked(R),C) :- server@CS,
        do(forward).
```
`locked(R)` *messages sent by a server are forwarded without further ado.*

Fig. 3. Law $\mathcal{TPL}$ ensuring serializability of transactions

how such designation provides one with the power implied by it. What concerns us here, given the sensitive role played by brokers in this community, is how can one designate only selected agents as brokers, not allowing anybody else to play this role. The problem is that such exclusivity cannot be ensured for implicit groups under LGI, as described so far. The reason for this is given below.

Since all members of an implicit community $\mathcal{C}$ start with identical control-state, the only way for an agent x to gain an exclusive status under the law of its community is as follows: x must be the only agent capable of having a certain term, such as `broker`, to be added to its control-state during its lifetime. But the addition of a new term to the CS of x, is, by definition, the consequence of some sequence of interactions between the members of a subgroup G of $\mathcal{C}$, which contains x. But if this is possible, then there can be nothing to prevent $\mathcal{C}$ to have another subgroup G', equivalent to G, which could add the term `broker` to the control-state of some agent x', violating exclusivity.

We now show how this equality-under-the-law of implicit communities under the present LGI can be broken by appealing to outside authorities via the well known concept of *certificates*.

3.1 The Role of Certificates in Distributed Systems

Computing over the Internet increasingly involves interaction between agents that are physically distant and have no knowledge of each others. As pointed out in [1], "such parties need to establish some trust in each other by receiving references from trusted intermediaries." Such intermediaries are often called *certifying authorities* (CAs), or simply *authorities*, and the references they produce are called *certificates*.

A certificate [8] is a four-tuple

$$\langle issuer, subject, attributes, signature \rangle$$

where, *issuer* is the public-key of the CA that issued and signed this certificate, *subject* is the public-key of the principal that is the subject of this certificate, *attributes* is what is being certified about the *subject*, and *signature* is the digital signature of this certificate by the *issuer*. Note that the *attributes* field is essentially a list of `(attribute, value)` pairs, represented here as a list of `attribute(value)` terms. For example, the attributes of a certificate might be the list `[name(johnDoe), role(manager)]`, asserting that the name of the subject in question is JohnDoe and his role in this community is a manager.

3.2 Using Certificates to Get Exclusive Privileges Under LGI

We now describe how a member of an $\mathcal{L}$-community $\mathcal{C}$ can obtain exclusive privileges and status by presenting a certificate issued by some outside authority. The degree of control over a given community thus provided to an authority outside of it is determined by the law of this community. That is, (a) it is the law of a community that determines the authorities whose certificates are acceptable to the community, and (b) the law determines the effect that a given certificate may have.

Submitting Certificates Consider an agent x that has a certificate c from an authority u. We have extended Moses with two Java methods that x can use for submitting a certificate to its controller $\mathcal{C}_x$.

The first method is for submitting what we call a *self-certificate*: a self-certificate submitted by x is one where the *subject* field is a public-key whose private counterpart is held by x itself. Such a certificate states something about its holder, such as the role he should be playing in a community. To submit such a certificate, x would use the following method:

```
sendSelfCertificate(c, sig),
```

where `c` is the certificate to be sent, presumably signed by some authority u, and `sig` is a digital signature generated by x itself, using the private counterpart of the public-key provided in the subject-field of c. This signature is used by the controller to validate that the agent it serves is in fact the subject of the presented certificate.

The second method is for the case where the subject of certificate c is *not* x himself. Such a certificate can be submitted to the controller via the method

```
sendCertificate(c),
```

requiring no digital signature by the sender.

Specifying Acceptable Certifying Authorities Of course, not every certifying authority would (nor should) be acceptable to a given community. Those that are acceptable, if any, can be specified in the law using clauses of the form:

```
authority(name, publicKey),
```

where `name` provides a convenient id for this authority within the law and `publicKey` provides a cryptographic identification for this authority to be used for verifying its signatures. Such clauses can be included in the *"preamble"* of the law, as in Figure 4. The set of all such clauses is called the *initial authority table* of the law. This is only an initial table, because LGI provides means for dynamically adding authorities to the authority table of an agent. (But space limitation prevents us from discussing these means and their use.)

Specifying the Effect of a Valid Certificate The submission of a certificate c by an agent x operating under law $\mathcal{L}$ to a controller $\mathcal{C}_x$ triggers the following sequence of events. First, an attempt is made to confirm that c is a valid certificate, duly signed by an authority that is acceptable to law $\mathcal{L}$, i.e., an authority that is represented in the authority table of the agent. An exception event is triggered if the certificate cannot be confirmed.

Second, if the confirmed c is a self-certificate, then an attempt is made to confirm the signature of x on it. An exception event is triggered if this signature is not confirmed.

Third, if no exception has been triggered thus far during the processing of c, then the following event would be triggered:

```
certified(X, certificate(issuer(I), subject(Y), attributes(A))),
```

where X is the agent who presented the certificate, I is the the local name (in the authority table under this law) of the issuer of the certificate, and A is the list of attributes of the certificates. As to parameter Y of this event, we need to distinguish between two cases: if c is a self-certificate, then Y is equal to X; otherwise, Y should be the public-key of the subject of this certificate.

Finally, what happens once the `certified` event is triggered depends entirely on the law in question. For example, in the case of law $\mathcal{TPL}'$ (see Figure 4), if an agent x sends a self-certificate

$$\langle issuer, subject, [role(broker)], signature\rangle$$

to his controller and if this certificate is duly signed by the certifying authority called `authority` in this law, then event:

```
certified(x, certificate(issuer(authority),

         subject(x), attributes([role(broker)]))),
```

will be triggered. Given Rule $\mathcal{R}8$, this event will result in the insertion of the term `broker` into x's control-state. The effect of such certification on a community is illustrated by the following refinement of our $\mathcal{TPL}$ law.

$\mathcal{TPL}'$: A Revision of the $\mathcal{TPL}$ Law A revision of the $\mathcal{TPL}$ law which supports brokers is displayed in Figure 4. This revision consist of (a) two `authority`-clauses, that define the initial authority table of the law; (b) replacement of Rules $\mathcal{R}1$, $\mathcal{R}2$ of $\mathcal{TPL}$ with Rule $\mathcal{R}1'$, $\mathcal{R}2'$; and (c) three new rules.

To understand this law, first note that by Rule $\mathcal{R}2'$, a lock request will be forwarded only if the destination server is recorded in the `serverList` term in the control-state of the requesting client. Now, every member starts with an empty `serverList` in its control-state. And only an agent having a term `broker` can add names of servers to this list (see Rule $\mathcal{R}9$ and $\mathcal{R}10$). This is what gives agents with a term `broker` their privileged role. The issue to be considered next is how does one appoint a specific agent to this role.

By Rule $\mathcal{R}8$, an agent can become a broker by presenting a certificate signed by one of the two authorities recognized by this law, which assert the right of the presenter to be a broker. Thus, one needs an authorization by a recognized authority to become a broker under this law. Similarly, an agent can act as a server only when presenting an appropriate certificate attesting its role (Rule $\mathcal{R}1'$).

4 Conclusion

Distributed computing on the Internet increasingly involves coordinating large groups of heterogeneous agents. This coordination is only possible if such agents have a credible basis for trusting each other. Under LGI such trust is established by imposing a single law over all members of the group. In this paper we extend the LGI mechanism by introducing a concept of *implicit groups*, or *communities*, *Preamble:*

authority(serverAuthority, publicKey).
authority(brokerAuthority, publicKey').
initialCS(serverList([])).

The authorities recognized by this law are: `serverAuthority` *and* `brokerAuthority`*.*
The initial control state of a member contains an empty server list.

$\mathcal{R}1'$.
```
certified(X,certificate(issuer(serverAuthority),subject(X),
      attributes([role(server)]))) :-
      do(+server).
```
If an agent presents a certificate signed by `serverAuthority` *asserting that it has the role of server, then the term* `server` *is added to its control state.*

$\mathcal{R}2'$.
```
sent(C,lock(R),S) :-
      serverList(SL)@CS, member(S,SL),
      !shrinking@CS,
      do(forward), do(+lock(R,S,pending)).
```
A message to lock resource `R` *is forwarded to its destination, only if the sender* `C` *is not in the shrinking phase, and if the destination is in the* `serverList` *of the sender.*

$\mathcal{R}8$.
```
certified(X,certificate(issuer(brokerAuthority),subject(X),
      attributes([role(broker)]))) :-
      do(+broker).
```
If an agent presents a certificate signed by `brokerAuthority` *asserting that it has the role of broker, then the term* `broker` *is added to its control state.*

$\mathcal{R}9$.
```
sent(B,assignServer(S),C) :- broker@CS,
      do(forward).
```
Only a broker can assign a server to a client `C`*.*

$\mathcal{R}10$.
```
arrived(B,assignServer(S),C) :-
      do(serverList(SL) <- serverList([S|SL])),
      do(deliver).
```
The arrival of the message `assignServer(S)` *to a client* `C` *causes the* `serverList` *of* `C` *to be appended with* `S`*.*

Fig. 4. $\mathcal{TPL}'$: a revision of the $\mathcal{TPL}$ law of Two-Phase-Locking

whose membership is left uncontrolled, and which require no central management of any kind. Specifically, we have discussed how an agent can join a community by adopting its law, how agents can name and locate each other, and how agents can use certificates and certifying authorities to acquire exclusive privileges.

References

1. A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 1000.

2. N.H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In David Garlan and Daniel Le Metayer, editors, *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 81–98, September 1997.
3. N.H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, May 1998.
4. N.H. Minsky and V. Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *7th USENIX Security Symposium*, January 1998.
5. N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 2000. (to be published, and currently available through `http://www.cs.rutgers.edu/~minsky/`).
6. N.H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. 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/`).
7. R. Rivest. The MD5 message digest algorithm. Technical report, MIT, April 1992. RFC 1320.
8. R. Rivest and B. Lampson. SDSI-a simple distributed security infrastructure. Technical report, MIT, 1996. http://theory.lcs.mit.edu/~rivest/sdsi.ps.
9. B. Schneier. *Applied Cryptography*. John Wiley and Sons, 1996.
10. A. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.