# Enforcement of Communal Policies for Peer-to-Peer Systems*

Mihail Ionescu, Naftaly Minsky, Thu D. Nguyen
{*ionescu, minsky, tdnguyen*}@*cs.rutgers.edu*

Technical Report DCS-TR-537
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ, 08854

December 2003

**Abstract.** *Peer-to-peer (P2P) computing, where peers in a community interact* directly *with each other rather than through intermediary servers, is emerging as a powerful paradigm for collaboration over the Internet. However, this paradigm poses a difficult challenge: how to ensure the harmonious, safe, and secure operation of these communities, particularly if they are large and geographically distributed, such that the collaborating principals may not even know or trust each other. Generally, members of such a community must conform to an application specific* communal policy *(or protocol), if the community is to operate smoothly and securely. Typically, the purpose of such a policy is either (a) to provide for effective coordination between members of the community, and/or (b) to ensure the security of community members, and of the information they share with each other.*

*The question we address in this paper is, how can such a communal policy be established reliably and in a scalable manner? That is, how can one ensure—in a manner consistent with the decentralized nature of the P2P model—that all members of a given P2P community comply with its communal policy? While some communities can rely on voluntary compliance with their stated policy, we believe that many policies required for future P2P applications will not lend themselves to voluntary compliance alone. Such policies, we maintain, need to be* enforced *to be reliable. We illustrate the nature of such policies by means of an example of a community that operates like Gnutella, but which is established to exchange more sensitive and critical information than music files.*

*Then, we propose to employ the intrinsically distributed control mechanism called* Law-Governed Interaction *(LGI) for the scalable enforcement of communal P2P policies. To demonstrate the efficacy of the proposed approach, we show how our example policy can be formulated and enforced under LGI. Finally, we modify an existing open-source Gnutella client to work with LGI and show that the use of LGI incurs little overhead.*

## 1 Introduction

Peer-to-peer (P2P) computing, where members of a community of agents interact *directly* with each other rather than through intermediary servers, is a potentially powerful paradigm for collaboration over the Internet. A member of a Gnutella [13] P2P community, for example, can locate a peer willing to provide a desired file via a communal flooding protocol that involves no central directory; and he can then get that file directly from the peer thus located. The advantages of such a community, when it operates effectively, are well known [23]. They include, (a) no single point of failure; (b) the ability of the community to grow almost without bounds; and (c) the ability to leave shareable resources to be maintained at their origin—rather than having to collect and maintain them in a central repository, incurring all the safety and reliability consequences of such centralization.

But a P2P community can collaborate harmoniously and securely only if all its members conform to a certain *communal policy*, or protocol. A Gnutella community, for example, relies on a flooding protocol to be carried out correctly by its members, in their collaborative effort to execute searches; and it depends on members not to issue too many queries, which might overburden the community, resulting in a kind of *tragedy of the commons*. Generally speaking, the purpose of such a policy is (a) to provide for effective coordination between members of the community, and (b) to ensure the security of community members, and of the information they share with each other. To achieve these purposes, the policy might impose constraints on both the membership of the community and on the behavior of its members when they are interacting with each other—all these in a highly application dependent manner.

*This work was supported in part by Panasonic Information and Networking Technologies Laboratory and by NSF grant No. CCR-98-03698.

**The question we address in this paper** is how can such a communal policy be established reliably and scalably? That is, how can one ensure—in a manner consistent with the decentralized nature of P2P communication—that all members of a given P2P community comply with its communal policy?

There are essentially two ways for establishing communal policies: by *voluntary compliance*, and by *enforcement*. A policy $P$ can be established by *voluntary compliance* as follows: once $P$ is declared as a standard for the community in question, each member is simply expected to abide by it, or to be carefully constructed according to it, or to use a widely available tool (or "middleware"), which is built to satisfy this policy. For example, to join a Gnutella community, one employs a Gnutella *servent*—several implementation of which are available—which is supposed to carry out the Gnutella flooding protocols for finding neighbors and information. This is entirely voluntary, and there is no way for a member to ensure, or verify, that its interlocutors use correct Gnutella servents.

However, for voluntary compliance to be reliable, the following two conditions need to be satisfied: (a) it must be in the vested interest of everybody to comply with the given policy $P$; and (b) a failure to comply with $P$, by some member, somewhere, should not cause any serious harm to anybody else. If condition (a) is not satisfied then there is little chance for $P$ to be generally observed. If condition (b) is not satisfied than one has to worry about someone not observing $P$, maliciously or inadvertently, and thus causing harm to others.

The boundary between communal policies that can, and cannot, be established by voluntary compliance is not sharp. There are, in particular, communities whose policies have been designed carefully to be resilient to some amount of violations; this is the case, for example, for the *Free Haven* [8] and *Freenet* [6] communities, which attempt to provide anonymity, and to prevent censorship. There are also communities whose activity is not important enough to worry about violations, even if conditions (a) and (b) above are not satisfied. A case in point is a Gnutella community when it is used to exchange music files. A small subset of members can overwhelm members connected by slow links, splintering the community, simply by issuing sufficiently large numbers of queries as actually happened at least once [7]. But exchange of music is not a critical activity for most people, so the risk of such a denial of service attack is not considered prohibitive. Also, contrary to condition (a) above, some people have an interest in pushing their own music, and they might provide it regardless of what has been requested. But such a violation of an implicit Gnutella policy can be easily detected, just by listening to what has been sent. No great harm is done to anybody by this violation, unless, perhaps, what has been sent is a virus.

But many of the policies required for P2P collaboration do not satisfy conditions (a) and (b) above, and do not lend themselves to implementation by voluntary compliance alone. Such policies, we maintain, *need to be enforced to be reliable*. We illustrate the nature of such policies, and their enforcement difficulty, by means of the following example of a community that operates like Gnutella, but which is established to exchange more critical information than music files.

**A Motivating Example:** Consider a collection of medical doctors, specializing in a certain area of medicine, who would like to share some of their experiences with each other—despite the fact that they may be dispersed throughout the world, and that they may not know each other personally. Suppose that they decided to form a Gnutella-like P2P community, to help in the location and dissemination of files that each of them would earmark for sharing with other community members.

The main difference between this case and music sharing is that the reliability of the information to be exchanged could be critical, and cannot always be judged simply by reading it. Moreover, some people, like representative of drug companies, may have vested interest to provide false treatment advice to promote their drug. One way to enhance the trustworthiness of the information exchanged is to limit the membership in the community to *trustworthy* people, however this may be defined. Policy *MDS* (for "Medical Data Sharing") below, attempts to do that, in a manner which is consistent with the decentralized nature of P2P communication. This policy also attempts to prevent the overloading of the community with too many messages, by budgeting the rate in which members can pose their queries; and it helps the receiver of information to evaluate its quality by providing him with the *reputation* of the source of each file he receives. Policy *MDS* is stated below, in somewhat abstract form.

1. Membership: An agent $x$ is allowed to join this community if it satisfies one of the following two conditions:

   (a) If $x$ is one of the *founders* of this community, as certified by a specific certification authority (CA) called here $ca1$. (Note: we assume that there are at least three such founders.)

   (b) (i) if $x$ is a medical doctor, as certified by the CA called $ca2$, representing the medical board; and
   (ii) if $x$ garners the support of at least three current members of this community.

And, a regular member (not a founder) is *removed* from this community if three different members vote for his removal.

2. Regulating the Rate of Queries: Every query has a cost, which must be paid for from the budget of the agent presenting it. (We will elaborate later on the cost of different queries, and on the way in which agents get their budgets.)

3. Reputation: Each member must maintain a *reputation value* that summarizes other members' feedback on the quality of his responses to posted queries. Further, this reputation must be presented along with every response to a query. (Again, we will elaborate on the exact mechanism for maintaining this reputation later.)

It should be clear that this policy cannot be entrusted to voluntary compliance. The question is: how can one enforce a single policy of this kind over a large and distributed community? A recent answer given to this question [11], in the context of distributed enterprise systems, is to adopt the traditional concept of *reference monitor* [3], which mediates all the interactions between the members of the community. Of course, a single reference monitor is inherently unscalable, since it constitutes a dangerous single point of failure, and could become a bottleneck if the system is large enough. This difficulty can be alleviated when dealing with static (stateless) policies, simply by replicating the reference monitor—as has been done recently in [14].

But replication is very problematic for dynamic policies, such as our *MDS*, which are sensitive to the history, or *state*, of the interactions being regulated. This is because every state change sensed by one replica needs to be propagated, synchronously, to all other replicas of the reference monitor. In the case of our *MDS* policy, in particular, all replicas would have to be informed synchronously about every request made by each member, lest this member circumvents his budget by routing different requests through different replicas. Such synchronous update of all replica could be very expensive, and is quite unscalable.

We propose in this paper to employ the intrinsically distributed control mechanism called *Law-Governed Interaction* (LGI) [18, 20] for the governance of P2P communities. LGI is outlined in Section 2. To demonstrate the efficacy of this approach for P2P computing, we show, in Section 3, how our example policy *MDS* can be formulated and enforced under LGI. This is carried out in a strictly decentralized and scalable manner, using the Gnutella file-sharing mechanism. Finally, to show that LGI imposes only a modest amount of overhead, we have modified an existing Gnutella servent to inter-operate with LGI. Section 5 presents the measured overhead, both in terms of increased latency for each message-exchange and overheads of running the LGI law enforcement middleware.

## 2 Law-Governed Interaction: An Overview

Broadly speaking, LGI is a message-exchange mechanism that allows an *open group* of distributed agents to engage in a mode of interaction *governed* by an explicitly specified policy, called the *law* of the group. The messages thus exchanged under a given law $\mathcal{L}$ are called $\mathcal{L}$-messages, and the group of agents interacting via $\mathcal{L}$-messages is called a *community* $\mathcal{C}$, or, more specifically, an $\mathcal{L}$-community $\mathcal{C}_{\mathcal{L}}$. The concept of LGI has been originally introduced (under a different name) in [18], and has been implemented via a middleware called Moses [20].

By the phrase "open group" we mean (a) that the membership of this group (or, community) can change dynamically, and can be very large; and (b) that the members of a given community can be heterogeneous. In fact, we make here no assumptions about the structure and behavior of the agents [1] that are members of a given community $\mathcal{C}_{\mathcal{L}}$, which might be software processes, written in an arbitrary languages, or human beings. All such members are treated as black boxes by LGI, which deals only with the interaction between them via $\mathcal{L}$-messages, making sure it conforms to the law of the community. (Note that members of a community are neither prohibited from non-LGI communication nor from participation in other LGI-communities.)

For each agent x in a given community $\mathcal{C}_{\mathcal{L}}$, LGI maintains what is called the *control-state* $\mathcal{CS}_x$ of this agent. These control-states, which can change dynamically, subject to law $\mathcal{L}$, enable the law to make distinctions between agents, and to be sensitive to dynamic changes in their state. The semantics of control-states for a given community is defined by its law and can represent such things as the role of an agent in this community, and privileges and tokens it carries. For example, under law $\mathcal{MDS}$ to be introduced in Section 3, as a formalization of our example *MDS* policy, the term

[1] Given the popular usages of the term "agent," it is important to point out that we do not imply by it either "intelligence" nor mobility, although neither of these is being ruled out by this model.

`member` in the control-state of an agent denotes that this agent has been certified as a doctor and admitted into the community.

We now elaborate on several aspects of LGI, focusing on (a) its concept of law, (b) its mechanism for law enforcement and their deployment, and (c) its treatment of digital certificates. Due to lack of space, we do not discuss here several important aspects of LGI, including the *interoperability* between communities and the treatment of *exceptions*. For these issues, and for a more complete presentation of the rest of LGI, and of its implemetation, the reader is referred to [20, 30, 4].

## 2.1 The Concept of Law

Generally speaking, the law of a community $\mathcal{C}$ is defined over a certain types of events occuring 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. The events subject to laws, called *regulated events*, include (among others): the *sending* and the *arrival* of an $\mathcal{L}$-message; the *coming due of an obligation* previously imposed on a given object; and the *submission of a digital certificate*. The operations that can be included in the ruling of the law for a given regulated event are called *primitive operations*. They include, operations on the control-state of the agent where the event occured (called, the "home agent"); operations on messages, such as `forward` and `deliver`; and the imposition of an obligation on the home agent.

Thus, a law $\mathcal{L}$ can regulate the exchange of messages between members of an $\mathcal{L}$-community, based on the control-state of the participants; and it can mandate various side effects of the message-exchange, such as 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 Nature 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 $\mathcal{C}$. This is due to the following properties of LGI laws:

- $\mathcal{L}$ 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 the same 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. And the locality of law enforcement enables LGI to scale with community size.

Finally, we note here that, as has been shown in [20], the use of strictly local laws does not involve any loss in expressive power. That is, any policy that can be implemented with a centralized reference monitor, which has the interaction state of the entire community available to it, can be implemented also under LGI—although the efficieny of the two types of implementation can vary.

**On the Structure and Formulation of Laws:** Abstractly speaking, the law of a community is a function that returns a *ruling* for any possible regulated event that might occur at any one of its members. The ruling returned by the law is a possibly empty sequence of primitive operations, which is to be carried out locally at the *home* of the event.

In the current implementation of LGI, a law can be written either in Prolog or Java. Under the Prolog implementation, which will be assumed in this paper, a law $\mathcal{L}$ 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`, is evaluated in the context of the control-state of this agent, producing the list of primitive-operations representing the ruling of the law for this event.

The four regulated events that are used in our law in Section 3 include `sent, arrived, obligationDue`, and `certified`. We will expand on obligations and how certificates are treated under LGI below. `sent` and `arrived` are defined as follows:

`sent(x, m, y):` a `sent` event occurs when an agent `x` sends an $\mathcal{L}$-message `m` addressed to another agent `y`. The sender `x` is considered the *home* of this event.

`arrived(x, m, y):` an `arrived` event occurs when an $\mathcal{L}$-message `m` sent by `x` arrives at `y`. The receiver `y` is considered the *home* of this event.

| 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; |
| **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 `y` |
| **Miscellaneous** | |
| `t@L` | returns true if term `t` is present in list L, and fails otherwise |
| imposeObligation (oType,dt) | causes the triggering of an `obligationDue(oType)` event after time interval `dt`. |

Figure 1: Some primitive operations in LGI.

In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals that have special roles to play in the interpretation of the law. These are the *sensor-goals*, which allow the law to "sense" the control-state of the home agent, and the *do-goals* that contribute to the ruling of the law. A *sensor-goal* has the form `t@CS`, where `t` is any Prolog term. It attempts to unify `t` with each term in the control-state of the home agent. A *do-goal* has the form `do(p)`, where `p` is one of the above mentioned primitive-operations. It appends the term `p` to the ruling of the law. A sample of primitive operations is presented in Figure 1.

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

Specifically, an obligation can be imposed on a given agent `x` at time `t0` by the execution at `x` of a primitive operation

```
imposeObligation(oType,dt)
```

where `dt` is the time period after which the obligation is to come due (`dt` is specified as a pair `[n,u]`, where `n` is an integer and `u` is a unit of time, such as "second" or "hour"), 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* before its due time $t=t_0+dt$, the *regulated event*

```
obligationDue(oType)
```

would occur at agent `x` at time `t`. The occurrence of this event would cause the controller to carry out the ruling of the law for this event; this ruling is, thus, the *sanction* for this obligation. Note that a pending obligation incurred by agent `x` can be *repealed* before its due time by means of the primitive operation

```
repealObligation(oType)
```

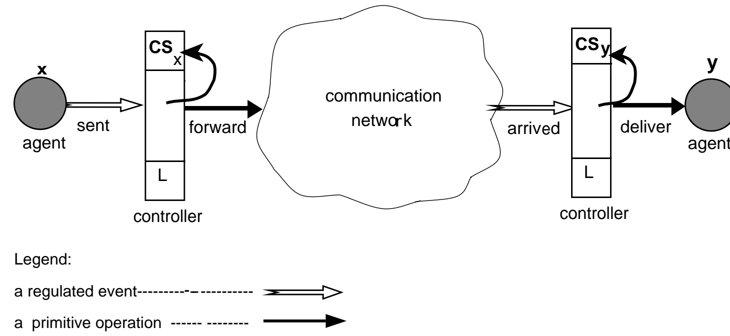



Figure 2: Law enforcement in LGI. Specifically, this figure shows the sequence of events and operations resulting from the sending of an $\mathcal{L}$-message from $x$ to $y$. When x initiates the send, a `sent` event is triggered at $\mathcal{T}_x$, which in turns causes a ruling at $\mathcal{T}_x$ that updates $\mathcal{CS}_x$ and forwards the message to $\mathcal{T}_y$. When the message arrives at $\mathcal{T}_y$, an `arrived` event is triggered, which in turns causes a ruling at $\mathcal{T}_y$ that updates $\mathcal{CS}_y$ and delivers the message to $y$.

carried out at `x`, as part of a ruling of some event. (This operation actually repeals *all* pending obligations of type `oType`).

For example, under law $\mathcal{MDS}$, (Figure 4), an obligation is imposed to grant each on-line member additional credit for querying the system every 60 seconds.

Note that there is a significant difference between this concept, and the concept of obligation under *deontic logic* [16], used for the specification of normative systems. The obligations of deontic logic allow one to reason about *what an agent must do*, but they provide no means for ensuring that what needs to be done will actually be done [25].

## 2.2 The Law-Enforcement Mechanism

We start with an observations about the term "enforcement," as used here. We do not propose to coerce any agent to exchange $\mathcal{L}$-messages under any given law $\mathcal{L}$. The role of enforcement here is merely to ensure that *any exchange of $\mathcal{L}$-messages, once undertaken, conforms to law $\mathcal{L}$*. More specifically, our enforcement mechanism is designed to ensure the following properties: (a) the sending and receiving of $\mathcal{L}$-messages conforms to law $\mathcal{L}$; and (b) a message received under law $\mathcal{L}$ has been sent under the same law (i.e., it is not possible to forge $\mathcal{L}$-messages).

Since we do not compel anybody to operate under any particular law, or to use LGI, for that matter, how can we be sure that all members of a community will adopt a given law? The answer is that an agent may be *effectively compelled* to exchange $\mathcal{L}$-messages, if he needs to use services provided only under this law, or to interact with agents operating under it. For instance, if the members of the medical information-sharing community posed as an example in Section 1 only accept $\mathcal{MDS}$-messages, then anybody wishing to participate in the community would be compelled to send $\mathcal{MDS}$-messages to them.

**Distributed Law-Enforcement:** Broadly speaking, the law $\mathcal{L}$ of community $\mathcal{C}$ is enforced by a set of trusted agents called *controllers*, that mediate the exchange of $\mathcal{L}$-messages between members of $\mathcal{C}$. Every member `x` of $\mathcal{C}$ has a controller $\mathcal{T}_x$ assigned to it ($\mathcal{T}$ here stands for "trusted agent") which maintains the control-state $\mathcal{CS}_x$ of its client `x`. And all these controllers, which are logically placed between the members of $\mathcal{C}$ and the communications medium (as illustrated in Figure 2) carry the *same law* $\mathcal{L}$. Every exchange between a pair of agents `x` and `y` is thus mediated by *their* controllers $\mathcal{T}_x$ and $\mathcal{T}_y$, so that this enforcement is inherently decentralized. Although several agents can share a single controller, if such sharing is desired. (The efficiency of this mechanism, and its scalability, are discussed in [20].)

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 any trusted machine, anywhere in the network. We have implemented a prototype *controller-service*, which maintains a set of active controllers. To be effective in a P2P setting, such a service would likely need to be well dispersed geographically so that it would be possible to find controllers that are reasonably close to their prospective clients.

**On the Basis for Trust Between Members of a Community:** For a members of an $\mathcal{L}$-community to trust its interlocutors to observe the same law, one needs the following assurances: (a) that the exchange of $\mathcal{L}$-messages is mediated

by controllers interpreting the *same law* $\mathcal{L}$; (b) that all these controllers are *correctly implemented*; and (c) all controllers run on trusted executing environments that will not maliciously cause them to misbehave even when they are correctly implemented. If these 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, that $\mathcal{L}$-messages cannot be forged.

To ensure that a message forwarded by a controller $\mathcal{T}_x$ under law $\mathcal{L}$ would be handled by another controller $\mathcal{T}_y$ operating under the *same* law, $\mathcal{T}_x$ appends a one-way hash [27] `H` of law $\mathcal{L}$ to the message it forwards to $\mathcal{T}_y$. $\mathcal{T}_y$ 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.

With respect to the correctness of the controllers and their execution environments, controllers must authenticate themselves to each other via certificates signed by a certification authority acceptable to the law $\mathcal{L}$. Note that different laws may, thus, require different certification levels for the controllers (and their execution environments) used for its enforcement. Messages sent across the network must be digitally signed by the sending controller, and the signature must be verified by the receiving controller. Such secure inter-controller interaction has been implemented in Moses ([19]).

**On the Deployment of LGI:** To deploy LGI, one needs a set of trustworthy controllers and a way for a prospective client to locate an available controller. While this requirement is reminiscent of the centralized reference monitor solution, there is a critical difference: *controllers can be distributed to limit the computing load placed on any one node, allow easy scaling, and avoid a single point of failure.* Further, due to the local enforceability of laws, each controller only has to mediate interactions that involve agents connected to itself.

One possible deployment strategy for P2P computing is to run controllers on a subset of the peers that are trusted by the community. A new member can query its peers for the addresses of these controllers. As the community grows, this subset of trusted peers can grow to accommodate the additional computing load. A second possible solution would be to use a controller service. An example of an analogous non-commercial service is the set of Mixmaster anonymous remailers [21].

## 2.3 The Treatment of Certificates under LGI

Under LGI, *all agents are made equal* at the time they join an $\mathcal{L}$-community. This is because the control-state of all new members is identical—and control-states provide the only means for a law to make distinctions between agents. We now explain how an agent can acquire extra privileges, thus becoming *more equal than others* (with apologies to George Orwell), by submitting appropriate certificates.

The submission by an agent $x$, operating under law $\mathcal{L}$, of a certificate `Cert` to its controller, has the following effect: an attempt is made to confirm that `Cert` is a valid certificate, duly signed by an authority that is acceptable to law $\mathcal{L}$, i.e., an authority that is represented by one of the `authority-clauses` in the preamble to the law (See Figure 3 for an example). If this attempt is successful[2], then a `certified` event is triggered. This event has as its argument the following representation of the submitted certificate

```
[issuer(I), subject(S), attributes(A)],
```

where `I` and `S` are internal representations of the public-keys of the CA that issued this certificate, and of its subject, respectively; `A` is what is being certified about the subject. Structurally, `A` is a list of `attribute(value)` terms. For example, the attributes of a certificate might be the list `[name(johnDoe), role(doctor)]`, asserting that the name of the subject in question is JohnDoe and his role in this community is a doctor. Additional components of the attributes field include the expiration time of the certificate, the URL of the server that maintains certificate revocation lists (CRLs) for this type of certificates, a certificate id (used to identify it in CRLs), etc. (Currently we support SPKI format of certificates [9].)

What happens when the `certified` event is triggered depends, of course, on the law. In the case of rule $\mathcal{R}2$ in law $\mathcal{MDS}$ (Figure 3), for example, the term `certified` is added to the control-state of the agent when it presents a doctor-certificate and triggers the certified event. This term reflects that the community now believes this agent to be a medical doctor.

[2] If the the certificate is found invalid then an *exception-event* is triggered.

# 3 Regulating Gnutella-Based Information Sharing: A Case Study

We now show how the policy *MDS* introduced in Section 1 can be explicitly specified and enforced using LGI. To provide a concrete context for our study, we assume that the community of doctors in question are sharing their medical data using Gnutella. We start this section with an outline of some aspects of the Gnutella protocol. We then introduce the law in three parts, the first part regulates membership, the second regulates the searches carried out by members, and the third part deals with the reputation of members. For each part, we first motivate the appropriate portion of policy *MDS*, then expand on the policy itself as necessary, and, finally, discuss the details of the law itself. We emphasize that this law is only an example; our central interest here is to establish the usability of a decentralized control system such as LGI rather than to write complex social laws/policies themselves.

Finally, before proceeding, we note that we did not choose Gnutella because it is necessarily the best P2P system. Rather, we choose Gnutella because: (a) it is a *real* protocol being used by a thriving community of users, (b) there are many different implementations of the Gnutella servent so that the community cannot rely on built-in safe-guards to enforce acceptable behavior, and (c) its protocol is relatively simple, ensuring that our study is not bogged down with many irrelevant details.

## 3.1 Gnutella: a Brief Overview

The Gnutella protocol is comprised of three main parts:

**Joining** An agent joins a Gnutella community by simply connecting (using TCP) to one or more active members.

**Peer discovery** After successfully contacting at least one active member, the joining agent uses a flooding *ping/pong* protocol to discover more active members (peers). This protocol is very similar to the search protocol that we will describe next.

**Search** An agent searches for shared content by flooding a query through the community. This flooding protocol works as follows. The querying agent sends its query along with a time-to-live (TTL) to a subset of its known peers. When a peer receives a query, (a) it decrements the TTL by one and forwards the query to all of its known peers if the TTL is still greater than 0; and (b) it searches its local store to see whether one or more files satisfy the query. If yes, then it sends a *query-hit* message containing the names of the matching files directly to the querying agent. A peer is allowed to drop a query, even if the TTL is non-zero, if it decides that the TTL is too large.

Finally, as the querying agent receives query-hit messages, it decides which file to download (if any) and does so by directly contacting the source of the appropriate query-hit message using HTTP.

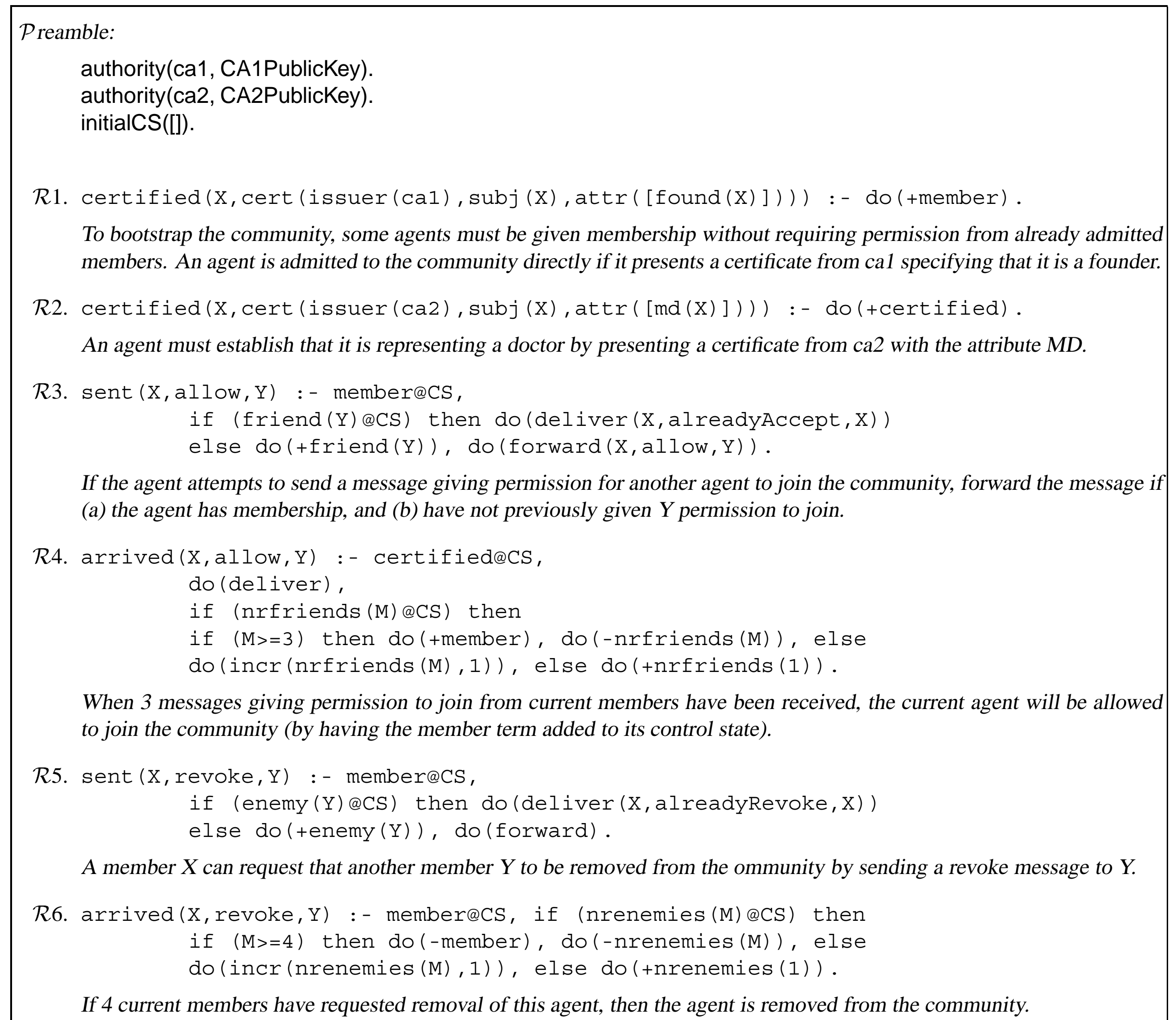
## 3.2 Regulating Membership

We start our case study by considering the problem of controlling membership. While the set of criteria for admission and removal can be arbitrarily complex, fundamentally, there seems to be three concerns for our supposed community: (a) members should be doctors since sensitive medical data should not be shared with non-doctors, (b) members should have some level of trust in each other—again, because of the sensitive nature of the information being shared, and (c) if a member misbehaves, it must be possible to revoke his membership. These requirements led us to the membership part of policy *MDS* presented in Section 1.

Figure 3 gives the first part of law $\mathcal{L}_{MDS}$, which implements the membership portion of policy *MDS*. This, like other LGI laws, is composed of two parts: a *preamble* and a set of *rules*. Each rule is followed by a comment (in italic), which, together with the explanation below, should be understandable even for a reader not well versed in the LGI language of laws (which is based on Prolog).

The preamble of law $\mathcal{L}_{MDS}$ has several clauses. The first two specify that this community is willing to accept certificates from two certifying authorities, `ca1` and `ca2`, identified by the given public keys. Then, there is an `initialCS` clause that defines the initial control-state of all agents in this community, which in this case is empty. We now examine the rules of this law in detail, showing how they establish the provisions of the policy at hand.

Rule $\mathcal{R}1$ allows the bootstrapping of a community by admitting three founding members certified by `ca1`. Rule $\mathcal{R}2$ allows an agent wishing to join the community to present a certificate from `ca2` to prove that its user is a doctor. Once certified as a founder, the term member will be inserted into the agent's control state.



*$\mathcal{P}$reamble:*

```
authority(ca1, CA1PublicKey).
authority(ca2, CA2PublicKey).
initialCS([]).
```

$\mathcal{R}1$.
```
certified(X,cert(issuer(ca1),subj(X),attr([found(X)]))) :- do(+member).
```
*To bootstrap the community, some agents must be given membership without requiring permission from already admitted members. An agent is admitted to the community directly if it presents a certificate from ca1 specifying that it is a founder.*

$\mathcal{R}2$.
```
certified(X,cert(issuer(ca2),subj(X),attr([md(X)]))) :- do(+certified).
```
*An agent must establish that it is representing a doctor by presenting a certificate from ca2 with the attribute MD.*

$\mathcal{R}3$.
```
sent(X,allow,Y) :- member@CS,
        if (friend(Y)@CS) then do(deliver(X,alreadyAccept,X))
        else do(+friend(Y)), do(forward(X,allow,Y)).
```
*If the agent attempts to send a message giving permission for another agent to join the community, forward the message if (a) the agent has membership, and (b) have not previously given Y permission to join.*

$\mathcal{R}4$.
```
arrived(X,allow,Y) :- certified@CS,
        do(deliver),
        if (nrfriends(M)@CS) then
        if (M>=3) then do(+member), do(-nrfriends(M)), else
        do(incr(nrfriends(M),1)), else do(+nrfriends(1)).
```
*When 3 messages giving permission to join from current members have been received, the current agent will be allowed to join the community (by having the member term added to its control state).*

$\mathcal{R}5$.
```
sent(X,revoke,Y) :- member@CS,
        if (enemy(Y)@CS) then do(deliver(X,alreadyRevoke,X))
        else do(+enemy(Y)), do(forward).
```
*A member X can request that another member Y to be removed from the ommunity by sending a revoke message to Y.*

$\mathcal{R}6$.
```
arrived(X,revoke,Y) :- member@CS, if (nrenemies(M)@CS) then
        if (M>=4) then do(-member), do(-nrenemies(M)), else
        do(incr(nrenemies(M),1)), else do(+nrenemies(1)).
```
*If 4 current members have requested removal of this agent, then the agent is removed from the community.*

Figure 3: $\mathcal{L}_{MDS}$, part 1: regulating membership.

Rule $\mathcal{R}3$ specifies that an agent wishing to sponsor the entry of a new member must already be a member of the community. Furthermore, this rule specifies that a member can only sponsor each distinct agent at most once. If this agent has previously sent permission to Y to join, return a message saying already accepted Y in the past. Rule $\mathcal{R}4$ specifies that if three members have approved the admittance of an agent, then that agent is admitted to the community as a new member. Each message granting permission is also delivered to the destination agent. Note the distributed enforcement of the approval process: for example, the controller of the approver ensures that it is a member while the controller of the agent being sponsored ensures that it is a certified doctor. This distributed regulation is important for scalability because the requesting agent's controller does not have to gather information about the granting agent; it simply knows that, according to the law, if it gets a reply granting permission, the proper conditions must have been checked and met at the granting agent's controller.

Finally, rules $\mathcal{R}5$ and $\mathcal{R}6$ regulate the removal of members from the community in a similar manner to how admittance is regulated.

## 3.3 Avoiding the Tragedy of the Commons

We now turn our attention to preventing members from abusing the community. In particular, we regulate the rate with which each member can make queries, to ensure that careless and/or selfish members cannot overwhelm the community. Before presenting the law, we expand on the synopsis of this part of policy *MDS* given in Section 1.

**Policy *MDS*, part (2): Limiting Query Rates**

2(a) Each agent has a query budget that starts at 500, and is incremented by 500 every minute up to a maximum of 50,000.

2(b) Each query has a cost as follows: $Cost = n \times 2^{TTL}$, where $n$ is the number of peers that the querying agent sends the query to and TTL is the query's TTL.

2(c) An agent is only allowed to pose a query if its budget is greater than the cost of the query. When the query is posed, the cost is deducted from the agent's query budget.

Note that even in the above expanded policy, we have chosen to ignore the issue of ensuring that agents faithfully follow the Gnutella flooding protocol. By ignoring this issue, we are ignoring a number of possible threats: for example, an agent might increase the TTL of a peer's query instead of decreasing it, using searches from other members as a method for denial-of-service attack on the community, or an agent might selectively refuse to forward queries from a subset of community members (perhaps because the owner of the agent doesn't like this subset of members). We have chosen not to address this issue, however, for two reasons: (1) for simplicity of presentation and to meet the space constraint; and (2) in our view, carelessness and/or selfish self-interest, as represented by someone trying to index the communal set of information [7], is a much more likely threat than a malicious insider attempting to perpetrate a denial-of-service attack.

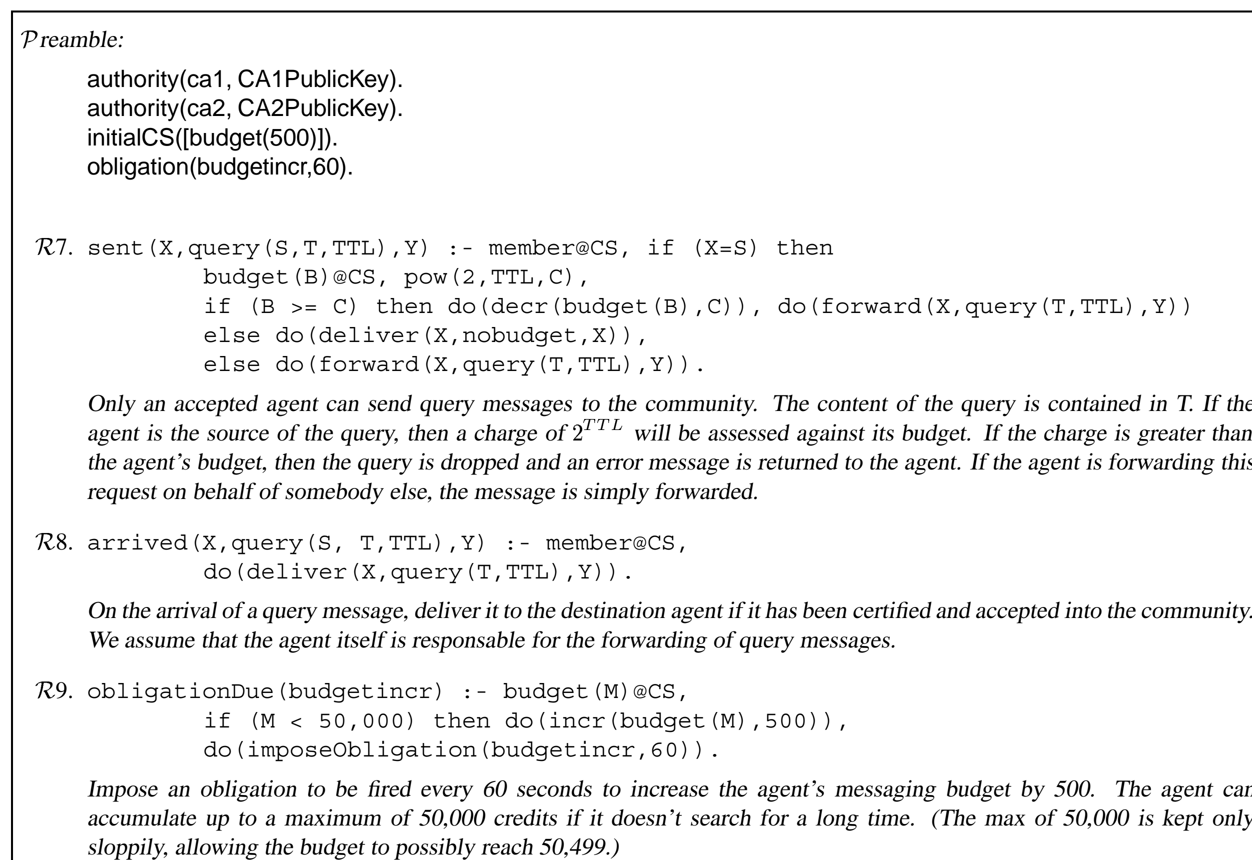
Figure 4 shows the second part of our law, which implements the second part of policy *MDS* described above. In Figure 4, the `preamble` is first modified to have a budget term that starts with 500 credits and an obligation imposed to be fired in 60 seconds. The firing of the obligation is handled by Rule $\mathcal{R}9$, which increases the agent's budget by 500 credits and then reimpose the obligation to be fired in another 60 seconds. Thus, effectively, the agent's budget is increased by 500 credits every minute until a cap of 50,000 is reached.

Rule $\mathcal{R}7$ deducts from the budget whenever the agent sends a query; if the agent does not have enough budget accumulated, the message is dropped and the agent notified with a message saying it has exceeded its budget. The specific values chosen were aimed at allowing a normal Gnutella search of $(TTL = 7, fan\text{-}out = 3)$ every minute and a maximal search of $(TTL = 14, fan\text{-}out = 3)$, which can be posted once every 100 minutes. Clearly, these parameters can be changed easily to accommodate the specific needs of the community. Also, note that we do not account for large fan outs once the query reaches an intermediary peer. We could handle this by complicating the law but do not do so here because there's no incentives for intermediary nodes to use huge fan outs (unless that node was malicious and wanted to perform a denial of service attack, a threat that we are explicitly not addressing).

Finally, note that we currently only keep a budget for query messages. Ping messages are costly as well. A community may or may not choose to regulate ping messages. The governance of these messages would be basically the same as that for the query messages so we do not consider them further here.

## 3.4 Maintaining Reputations

Finally, we develop a simple yet general reputation system to aid members in assessing the reliability of information provided by each others. A number of interesting reputation systems have already been designed [28, 2]. Most of these systems, however, have relied on a centralized server for computing and maintaining the reputations. Here, we show how a reputation system could be built using LGI's distributed enforcement mechanism. This reputation system is interesting because it maintains the reputation local to each peer (or, more precisely, local to the controller of that peer) so that the reputation can easily be embedded in every peer-to-peer query exchange without involving a third party. Further, our reputation system allows scalable, per-transaction, on-line updates of reputations. Note that while a peer's reputation is "local" to its controller, it cannot modify this reputation this is not allowed by our law, ensuring integrity of the reputation.



*$\mathcal{P}$reamble:*

```
authority(ca1, CA1PublicKey).
authority(ca2, CA2PublicKey).
initialCS([budget(500)]).
obligation(budgetincr,60).
```

$\mathcal{R}7$.
```
sent(X,query(S,T,TTL),Y) :- member@CS, if (X=S) then
        budget(B)@CS, pow(2,TTL,C),
        if (B >= C) then do(decr(budget(B),C)), do(forward(X,query(T,TTL),Y))
        else do(deliver(X,nobudget,X)),
        else do(forward(X,query(T,TTL),Y)).
```

*Only an accepted agent can send query messages to the community. The content of the query is contained in T. If the agent is the source of the query, then a charge of $2^{TTL}$ will be assessed against its budget. If the charge is greater than the agent's budget, then the query is dropped and an error message is returned to the agent. If the agent is forwarding this request on behalf of somebody else, the message is simply forwarded.*

$\mathcal{R}8$.
```
arrived(X,query(S, T,TTL),Y) :- member@CS,
        do(deliver(X,query(T,TTL),Y)).
```

*On the arrival of a query message, deliver it to the destination agent if it has been certified and accepted into the community. We assume that the agent itself is responsable for the forwarding of query messages.*

$\mathcal{R}9$.
```
obligationDue(budgetincr) :- budget(M)@CS,
        if (M < 50,000) then do(incr(budget(M),500)),
        do(imposeObligation(budgetincr,60)).
```

*Impose an obligation to be fired every 60 seconds to increase the agent's messaging budget by 500. The agent can accumulate up to a maximum of 50,000 credits if it doesn't search for a long time. (The max of 50,000 is kept only sloppily, allowing the budget to possibly reach 50,499.)*

Figure 4: $\mathcal{L}_{MDS}$, part 2: controlling querying behaviors.

Again, we first expand on the third part of policy *MDS* before presenting and describing the part of law $\mathcal{L}_{MDS}$ that implements this part of the policy.

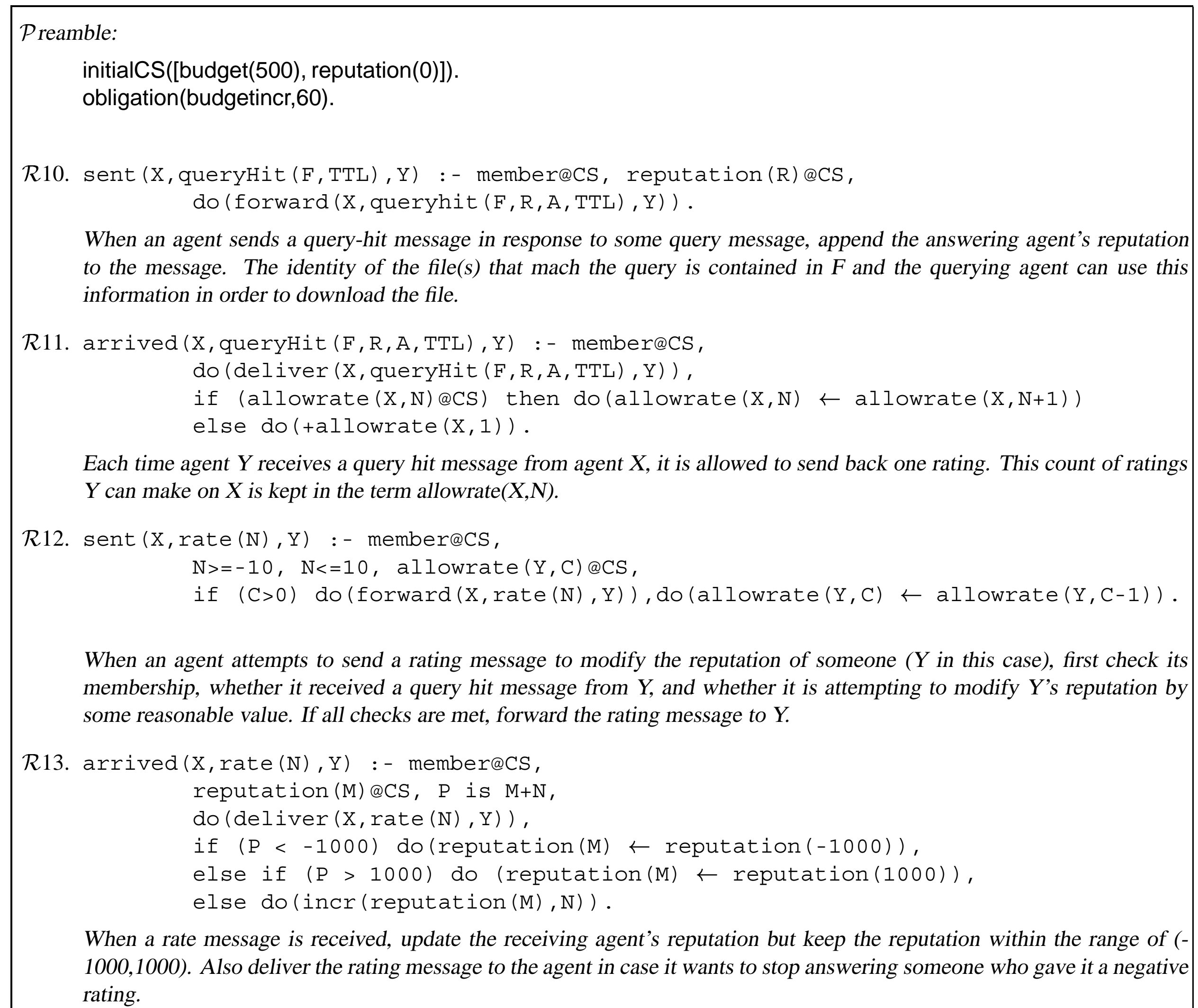
**Policy *MDS*, part 3: Reputation**

3(a) Each member of a community has a numeric reputation in the range of $-1000$ to $1000$, which is attached to every query-hit message so that the querier can decide whether to trust the answer or not. Larger numeric reputation values imply greater trust.

3(b) Each member starts with a reputation of 0 when it first joins the community. Whenever a member receives a query-hit message from a peer, it is allowed to rate the quality of the answer given by that peer. This rating can range from $-10$ to $10$ and is simply added to the peer's reputation. A member is allowed to make only one rating per answering peer per query.

While we have specified that the reputation is only attached to query-hit messages, one might imagine attaching it to query messages as well. Peers can then decide to answer queries based on the reputation of the querying agent. Also, one could easily weave more complex reputation policies if desired. For example, our original policy *MDS* also had an age component, where each member has an age reflecting the number of query-hit messages that it has sent. This age can helps peers to judge the reliability of each other's reputation. For example, I may choose to trust a new member with a reputation of 0 but not an old member with the same reputation: the older agent, even though it has been more active, hasn't garnered any trust over time, meaning that many of its answers must have been irrelevant to the posted query. We chose to remove this component to simplify our presentation ofthe law here.

Figure 5 gives the extensions needed to law $\mathcal{L}_{MDS}$, to implement this part of the policy. In Figure 5, we first modify the `Preamble` to have a reputation term, which starts at 0 when the agent first joins the community. Rules $\mathcal{R}10$ and $\mathcal{R}11$ regulate sending and arrival of query-hit messages: a member's reputation is added to each query-hit message.



$\mathcal{P}$*reamble:*

```
initialCS([budget(500), reputation(0)]).
obligation(budgetincr,60).
```

$\mathcal{R}$10.
```
sent(X,queryHit(F,TTL),Y) :- member@CS, reputation(R)@CS,
        do(forward(X,queryhit(F,R,A,TTL),Y)).
```

*When an agent sends a query-hit message in response to some query message, append the answering agent's reputation to the message. The identity of the file(s) that mach the query is contained in F and the querying agent can use this information in order to download the file.*

$\mathcal{R}$11.
```
arrived(X,queryHit(F,R,A,TTL),Y) :- member@CS,
        do(deliver(X,queryHit(F,R,A,TTL),Y)),
        if (allowrate(X,N)@CS) then do(allowrate(X,N) ← allowrate(X,N+1))
        else do(+allowrate(X,1)).
```

*Each time agent Y receives a query hit message from agent X, it is allowed to send back one rating. This count of ratings Y can make on X is kept in the term allowrate(X,N).*

$\mathcal{R}$12.
```
sent(X,rate(N),Y) :- member@CS,
        N>=-10, N<=10, allowrate(Y,C)@CS,
        if (C>0) do(forward(X,rate(N),Y)),do(allowrate(Y,C) ← allowrate(Y,C-1)).
```

*When an agent attempts to send a rating message to modify the reputation of someone (Y in this case), first check its membership, whether it received a query hit message from Y, and whether it is attempting to modify Y's reputation by some reasonable value. If all checks are met, forward the rating message to Y.*

$\mathcal{R}$13.
```
arrived(X,rate(N),Y) :- member@CS,
        reputation(M)@CS, P is M+N,
        do(deliver(X,rate(N),Y)),
        if (P < -1000) do(reputation(M) ← reputation(-1000)),
        else if (P > 1000) do (reputation(M) ← reputation(1000)),
        else do(incr(reputation(M),N)).
```

*When a rate message is received, update the receiving agent's reputation but keep the reputation within the range of (-1000,1000). Also deliver the rating message to the agent in case it wants to stop answering someone who gave it a negative rating.*

Figure 5: $\mathcal{L}_{MDS}$, part 3: maintaining reputations.

On receiving a query-hit message, the receiving agent can send back *one* rating (rule $\mathcal{R}$12). Note that LGI's distributed enforcement plays a role in shaping the implementation of the policy; in particular, the control state of each agent is used to maintain its reputation. Thus, to contribute to an agent's reputation, the rating agent can just send the rating message to the affected agent. The controller ensures that the agent cannot modify its reputation. Finally, when a rating message arrives, the controller modifies the affected agent's reputation accordingly. The reputation is constrained to stay between -1000 and 1000. A copy of the rating message is sent to the source agent of the query-hit message in case the agent decides to no longer answer peers that have given it a low rating.

# 4 Maintaining Persistent State

Peers in P2P networks often represent people; for example, in our example community, each member represents a doctor. Thus, persistent state such as membership and reputation should be associated with the person, not with the particular agent that he happens to be using at the moment. This is because as the real person moves around (between work and home, say), he may wish to employ different agents—the one installed on his work computer when at work and the one installed on his home computer when at home—at different locations. Thus, our system must be able to deal with the case where a peer disconnects and then reconnects later, potentially through a different controller.

There are three ways that such persistent state might be maintained and relocated across periods of disconnect: (1) force each agent to connect/reconnect at only one controller, (2) the community employs some centralized database to maintain this persistent state, or (3) have each agent ask a number of peers in the community to remember its persistent state. The first option is perhaps the simplest but is the most inflexible; if the agent is mobile, then it no longer has the option of connecting to the most convenient/efficient controller. Further, the state may still be lost if the controller crashes. Option two is also viable but requires the setup and maintenance of centralized resources. Such centralized resource is against the spirit of P2P communities and may limit scalability.

We explore here the third of these options, in a manner which is only outlined here, due to the limited space. One of the problems with carrieng out this option is that if an agent asks some small number of peers to hold its persistent state, what happens if these peers are themselves offline when the agent would like to rejoin the community? To address this problem, we introduce the notion of a *virtual agent* (VA), which was recently developed by Minsky and Ungureanu [17]. Under the current LGI model, a controller only create and maintain a control state for an agent when the agent contacts it to adopt a law. This model has been extended to allow an agent already operating under some law, to create a bare control-state for a non-existent agent, and can be used to deposit information in a tightly regulated manner.

We propose to employ VAs to maintain persistent state as follows. After an agent is admitted into the community, it is required to create a number of VAs. Once the agent has created its VAs, persistent information such as membership, reputation, and search credit can be sent to the VAs to keep on the behalf of the real agent. We will use `obligation` to regularly push any updates to the virtual agents. Then, whenever the agent disconnects and then rejoins, it simply asks the controller to retrieve its state from one of the VAs. Authentication is done through certificates or passwords.

We present in Figure 6 gives the fourth part of law $\mathcal{L}_{MDS}$, which implements the persistant state using the VA concept.

The agent can create one or more VAs as a persistent repository for its state, as shown in Rule $\mathcal{R}14$. In our current implementation this is an optional step for the agent, but it can be added as mandatory in the law (for example, after an agent is admitted it has to create at least one VA). The agent can then destroy the VA (Rule $\mathcal{R}15$) or ask for the state from a VA in Rule $\mathcal{R}16$. The actual state is obtained in Rule $\mathcal{R}17$ and will replace the current state. The state is exported to all of the VAs that act as permanent repositories at a constant interval of time (5 minutes currently), as shown in Rule $\mathcal{R}20$.

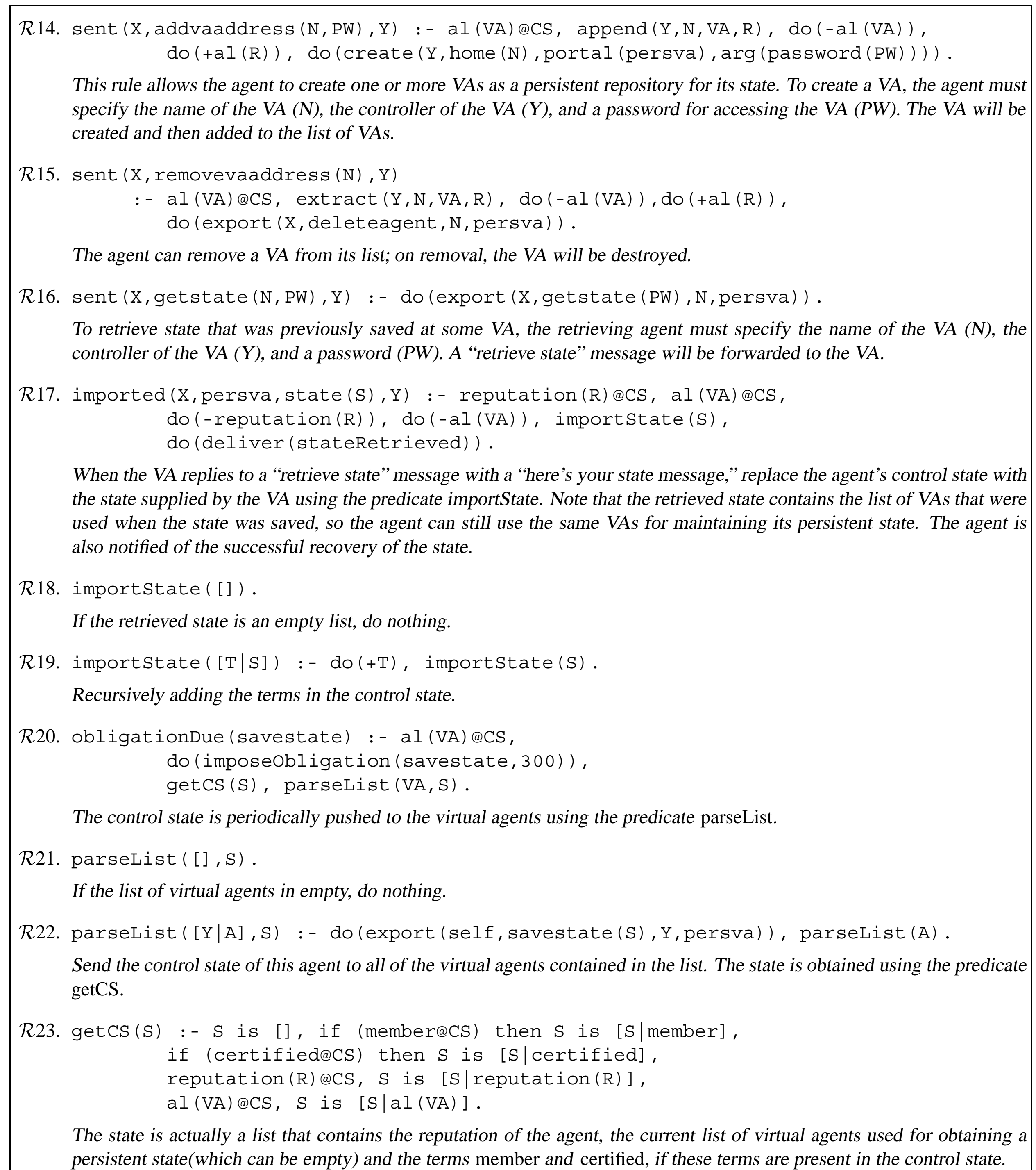
In Figure 7 we present the law for the virtual agent, $\mathcal{L}_{VA}$. The law implements two basic functions: saving and retrieving of the control state of an agent. The agent created the virtual agent with a specified password (by Rule $\mathcal{R}1$), that has to be checked when an attempt to obtain the state is made (Rule $\mathcal{R}3$).

Another issue that needs to be addressed is the following: what happens if a member of the community develops a bad reputation, disconnects and then reconnect without recovering his state. This would, of course, force the member to regain admittance by getting three approvals of admittance. This is possible if this dishonest member $c$ contacts three other members who do not know about his previous bad reputation. If $c$ is able to gain a new identity that is certified to be a doctor, then there's nothing we can do; we assume that it is difficult to gain a certificate from a trusted certifying authority with fake names. On the other hand, if $c$ attempts to regain membership with his previous identity, we can do the following. When an agent is first accepted in the community, a fingerprint (which is actually a large randomly generated number) is assigned to it by the controller. This fingerprint is added to the agent's control state and attached on to all outgoing messages from the agent by our law. Moreover, the fingerprint will also be persistently saved. Other agents can maintain a map between peers' identities and fingerprints. Then, if $c$ successfully regains membership without retrieving his earlier reputation, he will receive a new fingerprint. Eventually, if he interacts with any other peer that has a mapping from his identity to his old fingerprint, he will be discovered and can be expelled from the community at that point.

# 5 Performance

We have modified the Furi Gnutella agent [12] to work with LGI to show that it is practical to make software "LGI-aware" as well as to measure the imposed overheads. This LGI-aware (Furi-LGI) version preserves the full functionality of the original agent—it searches for files, keeps a list of active neighbors, etc.—but passes most messages (per law $\mathcal{L}_M DS$) through an LGI controller instead of sending directly to peers on the Gnutella network.

In this section, we assess the overhead of using LGI in two ways: (1) the added latency of passing through LGI's controllers for a pair of message exchange, and (2) the overhead of running a controller. The first is important



$\mathcal{R}$14.
```
sent(X,addvaaddress(N,PW),Y) :- al(VA)@CS, append(Y,N,VA,R), do(-al(VA)),
        do(+al(R)), do(create(Y,home(N),portal(persva),arg(password(PW)))).
```
*This rule allows the agent to create one or more VAs as a persistent repository for its state. To create a VA, the agent must specify the name of the VA (N), the controller of the VA (Y), and a password for accessing the VA (PW). The VA will be created and then added to the list of VAs.*

$\mathcal{R}$15.
```
sent(X,removevaaddress(N),Y)
     :- al(VA)@CS, extract(Y,N,VA,R), do(-al(VA)),do(+al(R)),
        do(export(X,deleteagent,N,persva)).
```
*The agent can remove a VA from its list; on removal, the VA will be destroyed.*

$\mathcal{R}$16.
```
sent(X,getstate(N,PW),Y) :- do(export(X,getstate(PW),N,persva)).
```
*To retrieve state that was previously saved at some VA, the retrieving agent must specify the name of the VA (N), the controller of the VA (Y), and a password (PW). A "retrieve state" message will be forwarded to the VA.*

$\mathcal{R}$17.
```
imported(X,persva,state(S),Y) :- reputation(R)@CS, al(VA)@CS,
        do(-reputation(R)), do(-al(VA)), importState(S),
        do(deliver(stateRetrieved)).
```
*When the VA replies to a "retrieve state" message with a "here's your state message," replace the agent's control state with the state supplied by the VA using the predicate importState. Note that the retrieved state contains the list of VAs that were used when the state was saved, so the agent can still use the same VAs for maintaining its persistent state. The agent is also notified of the successful recovery of the state.*

$\mathcal{R}$18.
```
importState([]).
```
*If the retrieved state is an empty list, do nothing.*

$\mathcal{R}$19.
```
importState([T|S]) :- do(+T), importState(S).
```
*Recursively adding the terms in the control state.*

$\mathcal{R}$20.
```
obligationDue(savestate) :- al(VA)@CS,
        do(imposeObligation(savestate,300)),
        getCS(S), parseList(VA,S).
```
*The control state is periodically pushed to the virtual agents using the predicate* parseList.

$\mathcal{R}$21.
```
parseList([],S).
```
*If the list of virtual agents in empty, do nothing.*

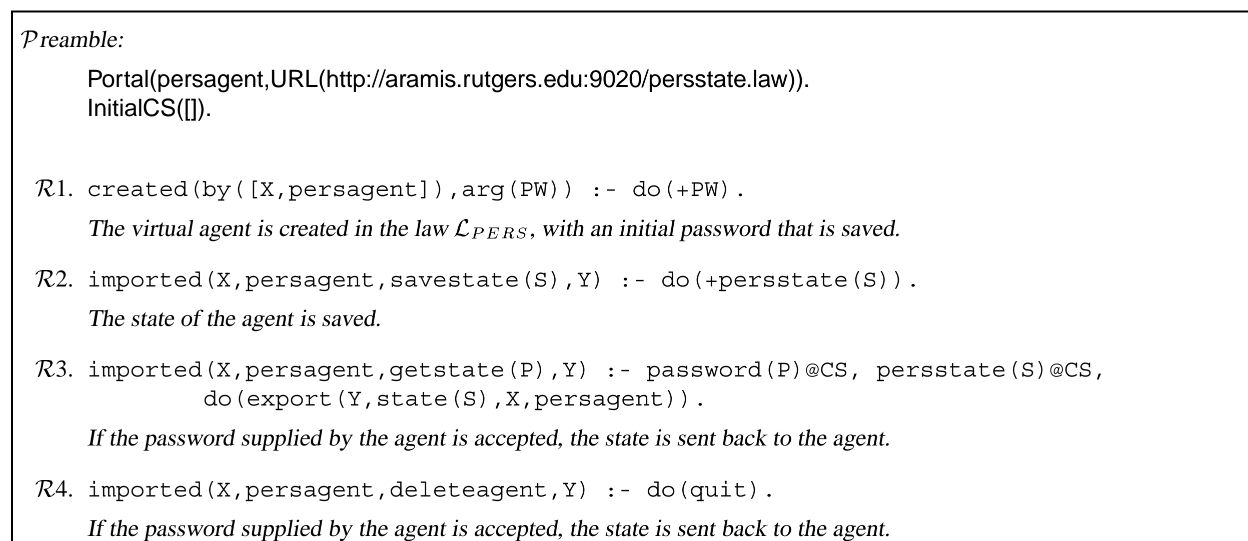
$\mathcal{R}$22.
```
parseList([Y|A],S) :- do(export(self,savestate(S),Y,persva)), parseList(A).
```
*Send the control state of this agent to all of the virtual agents contained in the list. The state is obtained using the predicate* getCS.

$\mathcal{R}$23.
```
getCS(S) :- S is [], if (member@CS) then S is [S|member],
        if (certified@CS) then S is [S|certified],
        reputation(R)@CS, S is [S|reputation(R)],
        al(VA)@CS, S is [S|al(VA)].
```
*The state is actually a list that contains the reputation of the agent, the current list of virtual agents used for obtaining a persistent state(which can be empty) and the terms* member *and* certified, *if these terms are present in the control state.*

Figure 6: $\mathcal{L}_{MD}$, part 4: maintaining persistent state

to demonstrate that LGI does not add so much latency as to make interactive P2P systems unusable. The latter is important in the case that controllers are run on the computing resources of a trusted subset of a community. Members will only be willing to host controllers if they impose relatively low overheads.

**Added Latency:** To measure the added latency of using LGI, we use two 440 MHz Sparc Ultra10 machines connected by an Ethernet hub, each hosting one Furi agent, and, in the LGI case, that agent's controller. Furi-LGI adopts our law



*Preamble:*

```
Portal(persagent,URL(http://aramis.rutgers.edu:9020/persstate.law)).
InitialCS([]).
```

$\mathcal{R}$1. `created(by([X,persagent]),arg(PW)) :- do(+PW).`

*The virtual agent is created in the law $\mathcal{L}_{PERS}$, with an initial password that is saved.*

$\mathcal{R}$2. `imported(X,persagent,savestate(S),Y) :- do(+persstate(S)).`

*The state of the agent is saved.*

$\mathcal{R}$3.
```
imported(X,persagent,getstate(P),Y) :- password(P)@CS, persstate(S)@CS,
        do(export(Y,state(S),X,persagent)).
```

*If the password supplied by the agent is accepted, the state is sent back to the agent.*

$\mathcal{R}$4. `imported(X,persagent,deleteagent,Y) :- do(quit).`

*If the password supplied by the agent is accepted, the state is sent back to the agent.*

Figure 7: $\mathcal{L}_{VA}$: The law governing the behavior of the VAs

$\mathcal{L}_{MDS}$ as presented in Section 3.

Results are as follows. A pair of Furi agents took 4ms for a pair of query/query-hit message exchange (2 messages). A pair of Furi-LGI agents, which requires 4 LGI evaluations and 4 additional inter-process but intra-machine messages, took 15ms. This is consistent with other microbenchmarks that we have run, which indicate that a typically LGI evaluation takes about 1-1.5ms. Of course, this time depends on the complexity of the law; however, our experience suggests that the incremental cost vs. complexity is small.

We observe that the overheads of LGI evaluations are unnoticeable to a human; a thousand evaluations would lead to little more than 1 second added latency. However, it is probably important for each peer to locate a controller that is relatively close by because each exchange of an $\mathcal{L}$-message requires two additional messages over a direct exchange, one from the source to its controller and one from the destination's controller to itself. (Although if the clients share a controller, then this reduces the overhead by one network delay since the controller-to-controller message is local.) In the Gnutella flood protocol, if peers are far away from their controllers, then the accumulated latency may become visible to the end user. However, if peers are relatively close to their controllers—say a message from a peer to its controller typically takes several tens of ms—then LGI's overheads are well within human tolerance. For example, a query/query-hit exchange that pass through a chain of 10 peers would require around 1 second.

**Cost of Running Controllers:** Measurements of a controller supporting our law $\mathcal{L}_{MDS}$ show that, on a relatively old 440 MHz Sparc Ultra10 machine, it can support approximately 35 constantly interacting clients at a cost of 20% of the CPU and 30MB of main memory. Again, while this cost is of course dependent on the complexity of the law, our experience suggests that the incremental cost vs. complexity is low. We expect that newer, faster desktops will be able to support significantly more clients while using only a small percentage of the processing power. Further, as clients are typically not continuously interactive, the above number is likely overly conservative.

# 6 Related Work

While current P2P systems such as Napster [22], Gnutella [13], and KaZaA [15] have been very successful for music and video sharing communities, little attention has been paid to community governance. The very fact that these communities are not regulating themselves has led to the illegal sharing of material and so much of the legal trouble between these communities and the entertainment industries.

Much recent work has studied the problem of how to better scale P2P systems to support an enormous number of users [32, 26, 29, 24]. These efforts have typically not concern themselves with security, however, which is the focus of our work.

Two recent works have considered how to implement reputation systems for P2P communities: Cornelli et. al have considered implementing a reputation-aware Gnutella servent [10] while Aberer and Despotovic have considered

implementing a binary trust system in the P-Grid infrastructure [1]. These efforts take a fundamentally different approach than ours in that they do not rely on a TCB as we do (i.e., the LGI controllers). Instead, they propose a system that uses historical information about past pair-wise interactions among the members of a community to compute a trust metric for each member. The underlying premise is that if only a few individuals are misbehaving, then it should be possible to identify these individuals by statistical analysis of the maintained historical information. Thus, these efforts differ from our work in two critical dimensions. First, these efforts limit their focus to reputation whereas our work uses reputation as a case study, aiming at the broader context of governing P2P communities. Second, these efforts rely on members of the community to participate in a reliable P2P storage and retrieval infrastructure while we rely on the network of controllers to provide a TCB.

A related effort that attempts to provide a more general framework than Cornelli et al. and Aberer and Despotovic is Chen and Yeager's Pablano web-of-trust [5]. While considerably more complex than the above two reputation systems, fundamentally, this work differs from our in similar ways.

Finally, it may be possible that some principles/invariants can be achieved even in the absence of universal conformance. One example of such a principle is anonymity: a number of P2P systems preserve the anonymity of their users by implementing anonymous storage and retrieval protocols that are resilient to a small number of misbehaving members [6, 31]. Our work differs from these efforts in that the LGI middleware allows a wide variety of policies to be specified and explicitly enforced without significant effort in protocol design, analysis, and implementation.

# 7 Conclusion

In this paper we propose LGI as the mechanism for specifying and enforcing the policies required for the members of a P2P community to collaborate harmoniously and securely with each other. We thus provide a solution for a critical open problem that must be addressed before P2P computing can realize its potential. LGI is well-suited to the P2P computing model because while it enforces *global* policies, it uses a *decentralized* enforcement mechanism that only depend on *local* information. This allows the use of LGI to easily scale with community sizes, avoids a single point of failure, and avoid needed centralized resources to start a community.

We provide supporting evidence for our proposal by presenting a case study; in particular, we applied LGI to a Gnutella-based information sharing community. We show how a law regulating important aspects of such communities, including membership, resource usage, and reputation control can be written in a straightforward manner in LGI. We also show how LGI's concept of a virtual agent can be used to persistently maintain state across multiple connections to the community. Finally, modifying a Gnutella agent to be compatible with LGI was a matter of a small number of weeks of work by one programmer.

We have also measured the overhead inherent in using LGI in governing a P2P community, from both the client and controller perspective. From the client's perspective, it is important that LGI does not increase the response time too much. Our measurements show that, if peers are not too far away from their controllers, a human should barely be able to tell that the Gnutella client would be running slower because of the indirection through LGI. Finally, from the controller's perspective, if some members of the community were to donate cycles from their personal machines to run the controllers for the community, it is important that this does not present too much of a computational overhead. Our measurements show that this overhead is quite low, allowing one controller to support tens (and perhaps hundreds on newer machines) of clients while using only a small part of the host's CPU and memory.

Finally, we observe that while our proposed solution requires the deployment of a trusted computing infrastructure, i.e., the set of law enforcement controllers, we believe that such an infrastructure is critical to the enforcement of policies that cannot rely on voluntary compliance. Further, this trusted computing infrastructure scales with the value of the collaboration being protected by the communal policy. Thus, a voted set of trusted nodes may be entirely adequate for the enforcement of many policies over many communities. Beyond this simple method, security may be escalated through a variety of approaches, such as the use of secure co-processors or a commercial controller service.

# References

[1] Karl Aberer and Zoran Despotovic. Managing Trust in a Peer-2-Peer Information System. In *Proceedings of the 10th International Conference on Inforamtion and Knowledge Management (ACM CIKM)*, 2001.

[2] The advogado project. Website: http://www.advogato.org/.

[3] J.P. Anderson. Computer security technology planning study. Technical Report TR-73-51, Air Force Electronic System Division., 1972.

[4] X. Ao, N. Minsky, T. Nguyen, and V. Ungureanu. Law-governed communities over the internet. In *Proc. of Fourth International Conference on Coordination Models and Languages; Limassol, Cyprus; LNCS 1906*.

[5] Rita Chen and William Yeager. Poblano: A Distributed Trust Model for Peer-to-Peer Networks. http://www.jxta.org/docs/trust.pdf.

[6] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in LNCS, pages 46–66, 2000.

[7] Clip2 DSS. Gnutella: To the Bandwidth Barrier and Beyond. http://www.clip2.com/gnutella.html, November 2000.

[8] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed Anonymous Storage Service. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in LNCS, pages 67–95, 2000.

[9] C. Ellison. The nature of a usable pki. *Computer Networks*, (31):823–830, November 1999.

[10] Sabrina De Capitani di Vimercati Stefano Paraboschi Fabrizio Cornelli, Ernesto Damiani and Pierangela Samarati. Implementing a Reputation-Aware Gnutella Servent. In *Proceedings of International Workshop on Peer to Peer Computing*, 2002.

[11] D. Ferraiolo, J. Barkley, and R. Kuhn. A role based access control model and refernce implementation within a corporate intranets. *ACM Transactions on Information and System Security*, 2(1), February 1999.

[12] The Furi Project. This project does not seem to be available on the web anymore. Please look for other projects based on Furi.

[13] Gnutella. http://gnutella.wego.com.

[14] G. Karjoth. The authorization service of tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001. (to appear).

[15] KaZaA. http://www.kazaa.com/.

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

[17] Naftaly Minsky and Victoria Ungureanu. Scalable Regulation of Inter-Enterprise Electronic Commerce. In *Proceedings of the Second International Workshop on Electronic Commerce*, 2001.

[18] N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.

[19] 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.

[20] 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*, 9(3):273–305, July 2000.

[21] Mixmaster. http://mixmaster.sourceforge.net.

[22] Napster. http://www.napster.com.

[23] Andy Oram, editor. *PEER-TO-PEER: Harnessing the Benefits of a Disruptive Technology*. O'Reilly & Associates, Inc., 2001.

[24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, 2001.

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

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

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

[28] The slashdot home page. Website: http://www.slashdot.org/.

[29] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[30] V. Ungureanu and N.H. Minsky. Establishing business rules for inter-enterprise electronic commerce. In *Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000); Toledo, Spain; LNCS 1914*.

[31] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A Robust, Tamper-Evident, Censorship-Resistant, Web Publishing System. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[32] Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, 2000.