Enforcement of Communal Policies for P2P Systems *

Mihail Ionescu, Naftaly Minsky, Thu D. Nguyen

Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ, 08854
{ionescu, minsky, tdnguyen}@cs.rutgers.edu

Abstract. We consider the question of how to establish and enforce *communal* policies for peer-to-peer (P2P) communities. Generally, members of each P2P community must conform to an application specific communal policy if the community is to operate smoothly and securely. An open question, however, is how can such communal policies be established reliably and in a scalable manner? While some communities can rely on voluntary compliance with their stated policies, voluntary compliance will not be sufficient for many future P2P applications. We illustrate the nature of policies that must be *enforced* to be reliable 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 [1]. 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.

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

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

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* [2] and *Freenet* [3] 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: the exchange of music is not a critical activity for most people and so the risk of members violating the implicit communal policy is not considered prohibitive.

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 en-

hance 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 [4], in the context of distributed enterprise systems, is to adopt the traditional concept of *reference monitor* [5], 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 [6].

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) [7, 8] 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 interoperate with LGI. Section 4 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 [7], and has been implemented via a middleware called Moses [8].

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 `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 [8–10].

[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.

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 [8], 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.

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.

The Concept of Enforced Obligation: Obligations, along with permissions and prohibitions, are widely considered essential for the specification of policies for financial enterprises [11]. The concept of obligation being employed for this purpose is usually based on conventional *deontic logic* [12], designed for the specification of normative systems, or on some elaborations of this logic, such as taking into account interacting agents [13]. These types of obligations allows 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 [14]. LGI, on the other hand, features a concept of obligation that can be enforced.

Informally speaking, an obligation under LGI is a kind of *motive force*. Once an obligation is imposed on an agent—which can be done as part of the ruling of the law for some event 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 community. For a detailed discussion of obligations the reader is referred to [8].

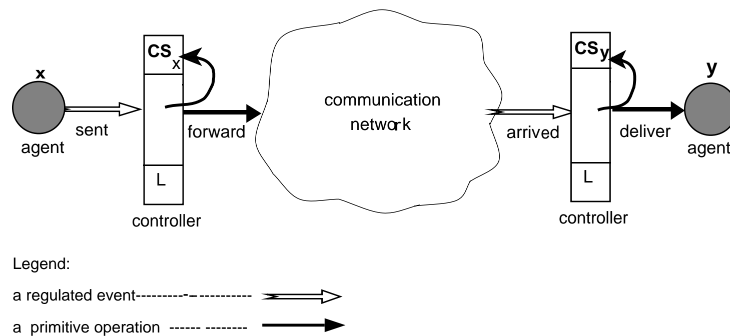

Fig. 1. Law enforcement in LGI. Specifically, this figure shows the sequence of events and operations resulting from the sending of an $\mathcal{L}$-message from x to y. When x initiates the send, a `sent` event is triggered at $\mathcal{T}_x$, which in turns causes a ruling at $\mathcal{T}_x$ that updates $\mathcal{CS}_x$ 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.

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 1) 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 [8].)

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 [15] 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 ([16]).

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 [17].

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 the critical 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—although we do not present the last part of the law, which deals with reputation, because of space constraints; we instead refer the interested reader to [18].

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. In particular, 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 and forwards the query to a subset of its known peers if the TTL is greater than 0; and (b) it searches its local store for files matching the query. If relevant files are found, it sends a *query-hit* message containing the names of the matching files directly to the querying agent. 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 2 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.

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

$\mathcal{P}$*reamble:*

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

$\mathcal{R}$1.

```
certified(X,cert(issuer(ca1),subj(X),attr([found(X)])))
   :- do(+member).
```

The agent is accepted into the community without requiring approval from already admitted members if it can present a certificate from ca1 stating that it is a founder.

$\mathcal{R}$2.

```
certified(X,cert(issuer(ca2),subj(X),attr([md(X)])))
   :- do(+certified).
```

The 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 approving admittance of another agent, 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 approval messages have been received, the 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).
```

The agent (X) can request that another agent (Y) to be removed from the community 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.

Fig. 2. $\mathcal{L}_{MDS}$, part 1: regulating membership.

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 [19], is a much more likely threat than a malicious insider attempting to perpetrate a denial-of-service attack.

Figure 3 shows the second part of our law, which implements the second part of policy *MDS* described above. In Figure 3, 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 ($TTL = 7, \textit{fan-out} = 3$) every minute and a maximal search of ($TTL = 14, \textit{fan-out} = 3$), 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

$\mathcal{P}$*reamble:*

```
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.*

$\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.

Fig. 3. $\mathcal{L}_{MDS}$, part 2: controlling querying behaviors.

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

The last part of our case study involves a simple yet general reputation system to aid members in assessing the reliability of information provided by each others. We do not include the law itself because of space constraints. Rather, we expand on the third part of policy *MDS* to give the general idea of the reputation system and then refer the interested reader to [18] for the details of the law.

A number of interesting reputation systems have already been designed [20, 21]. 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 be-

cause 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. This allows scalable, per-transaction, on-line updates of reputations.

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.

3.5 Maintaining Persistent State

To close the section, we observe that 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. We use a feature of LGI called *virtual agents* [22] to provide the needed persistency. Again, we refer the interested reader to [18] for the details of this implementation.

4 Performance

We have modified the Furi Gnutella agent [23] 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.

To give an idea of the cost of using LGI, we measured two aspects of Furi-LGI: (1) the added latency of passing through LGI's controllers for a pair of message exchange, and (2) the overhead of running a controller. Measurements were taken on two 440 MHz Sparc Ultra10 machines connected by an Ethernet hub. Results are as follows. A query/query-hit message exchange between two normal Furi agents took 4ms compared to 15ms for two Furi-LGI agents. The added overhead for Furi-LGI is due to 4 LGI evaluations and 4 additional inter-process but intra-machine messages. A controller supporting 35 constantly interacting clients used on average 20% of the CPU and 30MB of main memory. While we did not measure the system extensively, these results

provide evidence that LGI does not impose undue overheads and so provides a practical vehicle for establishing communal policies for P2P systems.

5 Related Work

While current P2P systems such as Gnutella [24], and KaZaA [25] 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 [26–29]. 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 [30] while Aberer and Despotovic have considered implementing a binary trust system in the P-Grid infrastructure [31]. 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 [32]. 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 [3, 33]. 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.

6 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. 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 of 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. Further, modifying a Gnutella agent to be compatible with LGI was a matter of a small number of weeks of work by one programmer.

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. Oram, A., ed.: PEER-TO-PEER: Harnessing the Benefits of a Disruptive Technology. O'Reilly & Associates, Inc. (2001)
2. Dingledine, R., Freedman, M.J., Molnar, D.: 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 (2000) 67–95
3. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: 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 (2000) 46–66
4. Ferraiolo, D., Barkley, J., Kuhn, R.: A role based access control model and refernce implementation within a corporate intranets. ACM Transactions on Information and System Security **2** (1999)
5. Anderson, J.: Computer security technology planning study. Technical Report TR-73-51, Air Force Electronic System Division. (1972)
6. Karjoth, G.: The authorization service of tivoli policy director. In: Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001). (2001) (to appear).
7. Minsky, N.: The imposition of protocols over open distributed systems. IEEE Transactions on Software Engineering (1991)
8. Minsky, N., Ungureanu, V.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. TOSEM, ACM Transactions on Software Engineering and Methodology **9** (2000) 273–305
9. Ungureanu, V., Minsky, N.: Establishing business rules for inter-enterprise electronic commerce. (In: Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000); Toledo, Spain; LNCS 1914)
10. Ao, X., Minsky, N., Nguyen, T., Ungureanu, V.: Law-governed communities over the internet. (In: Proc. of Fourth International Conference on Coordination Models and Languages; Limassol, Cyprus; LNCS 1906)

11. Linington, P.: Options for expressing ODP enterprise communities and their policies by using UML. In: Proceedings of the Third Internantional Enterprise Distributed Object Computing (EDOC99) Conference, IEEE (1999)
12. Meyer, J.J.C., Wieringa, R.J., F.P.M., D.: The role of deontic logic in the specification of information systems. In Chomicki, J., Saake, G., eds.: Logic for Databases and Information Systems. Kluwer (1998)
13. Brown, M.: Agents with changing and conflicting commitments: a preliminary study. In: Proc. of Fourth International Conference on Deontic Logic in Computer Science (DEON'98). (1998)
14. Linington, P., Milosevic, Z., Raymond, K.: Policies in communities: Extending the odb enterprise viewpoint. In: Proceedings of the Second Internantional Enterprise Distributed Object Computing (EDOC98) Conference, IEEE (1998)
15. Schneier, B.: Applied Cryptography. John Wiley and Sons (1996)
16. Minsky, N., Ungureanu, V.: A mechanism for establishing policies for electronic commerce. In: The 18th International Conference on Distributed Computing Systems (ICDCS). (1998) 322–331
17. Mixmaster. (http://mixmaster.sourceforge.net)
18. Ionescu, M., Minsky, N., Nguyen, T.: Enforcement of communal policies for p2p systems. Technical Report DCS-TR-537, Department of Computer Science, Rutgers University (2003)
19. Clip2 DSS: Gnutella: To the Bandwidth Barrier and Beyond. http://www.clip2.com/gnutella.html (2000)
20. The SlashDot Home Page. (Website: http://www.slashdot.org/)
21. Advogado. (Website: http://www.advogato.org/)
22. Minsky, N., Ungureanu, V.: Scalable Regulation of Inter-Enterprise Electronic Commerce. In: Proceedings of the Second International Workshop on Electronic Commerce. (2001)
23. The Furi Project. (Website: http://www.furi.org)
24. Gnutella. (http://gnutella.wego.com)
25. KaZaA. (http://www.kazaa.com/)
26. Zhao, Y., Kubiatowicz, J., Joseph, A.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley (2000)
27. Rowstron, A., Druschel, P.: 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)
28. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: Proceedings of the ACM SIGCOMM '01 Conference. (2001)
29. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings of the ACM SIGCOMM '01 Conference. (2001)
30. Fabrizio Cornelli, Ernesto Damiani, S.D.C.d.V.S.P., Samarati, P.: Implementing a Reputation-Aware Gnutella Servent. In: Proceedings of International Workshop on Peer to Peer Computing. (2002)
31. Aberer, K., Despotovic, Z.: Managing Trust in a Peer-2-Peer Information System. In: Proceedings of the 10th International Conference on Information and Knowledge Management (ACM CIKM). (2001)
32. Chen, R., Yeager, W.: Poblano: A Distributed Trust Model for Peer-to-Peer Networks. (http://www.jxta.org/docs/trust.pdf)
33. Waldman, M., Rubin, A.D., Cranor, L.F.: Publius: A Robust, Tamper-Evident, Censorship-Resistant, Web Publishing System. In: Proceedings of the 9th USENIX Security Symposium. (2000)