In Vivo Evolution of Policies that Govern a Distributed System

Constantin Serban
Applied Research
Telcordia Technologies
serban@research.telcordia.com

Naftaly Minsky
Department of Computer Science
Rutgers University
minsky@cs.rutgers.edu

Abstract

This paper addresses an important open problem confronting any decentralized and stateful access control (AC) mechanism for networked systems, particularly when the system at hand is large, heterogeneous and open. The problem, in a nutshell, is how to enable safe evolution of the policy that governs a given system, while that system continues to operate. This problem, and its solution, are addressed here in terms of the Law-Governed Interaction *(LGI) mechanism, but the ideas underlying this paper should be broadly relevant to other decentralized and stateful control mechanisms, such as the use of distributed firewalls for the protection of distributed enterprise systems.*

1 Introduction

This paper addresses an important open problem confronting any decentralized and stateful access control (AC) mechanism for networked systems, particularly when the system at hand is large, heterogeneous and open. The problem, in a nutshell, is how to enable safe evolution of the policy that governs a given system, while that system continues to operate. The need for such an *in vivo* evolution (i.e., evolution in a living organism, as it where) of system's policy is self evident, when dealing with a long lived system that must operate continuously. But as we shall see, such evolution can be very harmful to the integrity of the system being governed by the evolving policy. (Consider, for analogy, the changing of traffic law in London to the European law, in the middle of rush hour.) This problem, and its solution, are discussed here in terms of the governance mechanism called *Law-Governed Interaction* (LGI), but the ideas underlying this paper should be broadly relevant to other decentralized and stateful control mechanisms, such as the use of distributed firewalls for the protection of enterprise systems.

The rest of this section is organized as follows. We first provide a very brief introduction of the LGI mechanism, which should be sufficient for this paper (for more about LGI the reader is referred to the manual that accompany its release [9]). Then, in Section 1.2, we introduce a model for *in vivo* evolving laws for distributed systems. And in Section 1.3 we discuss the challenges posed by such evolution.

1.1 A Very Brief Introduction to LGI

LGI is a generalized AC mechanism that enables a group of distributed actors—collectively called here a *system*—to exchange messages subject to a *law* that governs them all, and which is enforced securely and in a decentralized manner.

More specifically, an actor A_x that operates under a given law $\mathcal{L}$ exchanges messages with the rest of the system via a proxy, called the controller, trusted to enforce law $\mathcal{L}$. A controller is a triple $T_x^{\mathcal{L}} = \langle \mathcal{L}, I, CS_x \rangle$, where $\mathcal{L}$ is the law under which this particular actor operates; CS_x is the *control-state* (or simply *state*) maintained by the controller on behalf of A_x; and I is a generic mechanism that mediates the interactions of x with others, according to the given law $\mathcal{L}$. The pair consisting of an actor and its controller is called an agent $x = \langle A_x, T_x^{\mathcal{L}} \rangle$. Figure 1 shows the passage of a message from an actor A_x to A_y, as it is mediated by a pair of controllers, first by $T_x^{\mathcal{L}}$, and then by $T_y^{\mathcal{L}}$.

The purpose of a law under LGI is to decide what should be done in response to the occurrence of certain *regulated events*—such as the receipt or the sending of a message at the controller. For any such event e occurring at an agent x (i.e., at the controller), the law mandates a response to be carried out at x. This mandated response, called the *ruling* of the law, is a function $L(e, s)$, where s is the state of agent x at the time of the occurrence of event e. Such a ruling is a sequence of zero or more *control operations*, which can cause such things as forwarding of messages, and updating the state s of the home agent x. (Note that the ruling of the law is not limited to accepting or rejecting a message, as under most conventional AC mechanism. This fact is a source of much of the power of LGI.)

It is worth noting that that LGI replaces the conventional concept of *policy* under AC—traditionally defined as an *ac-*

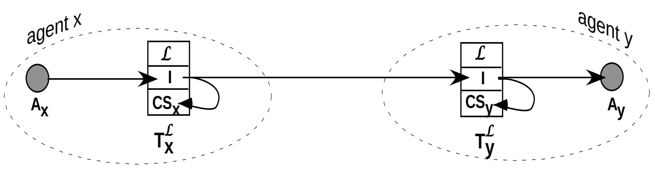

Figure 1. *A pair of interacting $\mathcal{L}$-agents*

cess control matrix—with the pair $\langle law, state\rangle$, where the fixed law holds sway over an entire system, while the state is a distributed collection of local states, each of which can change dynamically independently of the others, but subject to the global law.

LGI laws can be specified using different languages, without changing the semantics of the mechanism. Under the current implementation of LGI, in particular, one can choose between two such languages, based on Prolog and Java, respectively. The examples of laws in this paper are specified by means of an informal pseudo-code under which a law is a sequence of event-condition-action rules of the following form: $upon\langle event\rangle\ if\langle condition\rangle\ do\langle action\rangle$ where the $\langle event\rangle$ represents one of the regulated events; the $\langle condition\rangle$ is a general expression defined over the event and the state of the agent; and the $\langle action\rangle$ is one or more operations mandated by the law.

Finally, we introduce here briefly two features of LGI, called *obligations* and *exceptions*, which are required due to the decentralized and stateful nature of this mechanism, and which are relevant to this paper. (For more about these features, and for their rationale, see [9]). First, the ruling of an LGI law may impose an *obligation* on the agent, causing a specific event to be triggered sometime in the future, after a specified period of time, thus providing a *proactive capability*. Second, an event called an `exception`, can be triggered when an operation previously mandated by the law cannot be completed successfully—and it is up to the law to prescribe a recovery measure. We will see examples of the use of these features in the following section.

1.2 A Model of *In Vivo* Evolving Laws

We assume that a law of a given distributed system S undergoes a process of evolution through a lineage

$$\overline{\mathcal{L}} = \langle \mathcal{L}_0, \mathcal{L}_1, ..., \mathcal{L}_i, ..., \mathcal{L}_n = \mathcal{L}_c\rangle$$

of law versions. Law $\mathcal{L}_n$, the latest of these versions, at a given moment in time, is by definition the current law of system S—it is also denoted by $\mathcal{L}_c$.

Ideally, one would like all agents of system S to always operate under the current law $\mathcal{L}_c$ of its lineage $\overline{\mathcal{L}}$. That is, when a new version of the lineage is introduced, one would like everybody in the system to switch to it immediately, and in a virtually atomic manner. This is readily doable under a centralized AC mechanism that uses a single reference monitor to mediates all message exchanges in the system. But as we shall see below, under a decentralized mechanism, such as LGI, this ideal can only be achieved under some stringent conditions, and over a considerable period of time. Therefore one needs to employ a more incremental technique for converging to a state in which all members of S operate under the same law $\mathcal{L}_c$. This implies that during the convergence period different agents that interact with each other might be operating under different versions of the law in lineage $\overline{\mathcal{L}}$. We call this situation the *dispersion* of the law.

The fundamental reason for such dispersion is that in order for the system as a whole to adopt the new $\mathcal{L}_c$ one needs to notify all the agents of this system about the new current law; and have each of them, individually, change its own law to the new $\mathcal{L}_c$. Moreover, these law changes need to be synchnronized, and be carried out in a virtually atomic manner. But this is hard to achieve, for several reasons.

First, the composition of the system at hand may not be fully known. Because the system may be open, allowing for new agents to be formed, and for existing ones to disappear, with no centralized control and without registration[1]. Therefore the system administrator (or the agent responsible for introducing the new law version $\mathcal{L}_c$) may not have enough knowledge to notify all system agents about the new version. Second, even if the composition of S is known to the administrator, some of the members of S may be temporarily unreachable, due to a network failure, or because they are not online. Finally, some of the agents that are notified of the new $\mathcal{L}_c$, may not be able or willing to change their law to it immediately, because they are in the middle of some transaction that needs to be concluded under the previous law.

Therefore, the system may have to operate under a dispersed lineage $\overline{\mathcal{L}}$, during certain periods in the course of its evolution. During such a period of dispersion we sometime refer to $\overline{\mathcal{L}}$ as "the law of the system". These periods of dispersion may be transient, but they may also be quite lengthy if the speed of evolution of the law exceeds the speed of convergence to new versions of it.

It is worth introducing at this point the following notations. An arbitrary law version in a lineage $\overline{\mathcal{L}}$ is denoted simply as $\mathcal{L}$. The immediately preceding and succeeding version of $\mathcal{L}$ in $\overline{\mathcal{L}}$ are denoted by $\mathcal{L}^-$ and $\mathcal{L}^+$, respectively; an arbitrary version to the left or right of $\mathcal{L}$ in the lineage $\overline{\mathcal{L}}$ is denoted by $\mathcal{L}^{-*}$ and $\mathcal{L}^{+*}$, respectively. Finally, an agent x operating under a law $\mathcal{L}$ will be denoted by $x/\mathcal{L}$.

[1]Although LGI can establish central control and registration via appropriate laws, it does not require it; and it is unnecessary in many applications.

1.3 On the Challenges Posed by *In Vivo* Evolution of Laws Under Decentralized AC

While *in vivo* evolution of the law that governs a distributed system is often necessary, it can be perilous to the integrity of the system, and must be handled with care. The following are the main problems confronting such evolution, when the laws are stateful, and when they are enforced in a decentralized manner.

- When a new version of the law is introduced, it is not enough to replace the old version of the law of each member of S with the new one. One also needs to update the state of each agent to fit the new law, thus preserving, under the new law, the meaning of things like roles, capabilities, or status of ongoing transactions, which are represented in the state of individual agents. Accordingly, the adaptation of the state to the new law should be done on a per-agent basis. The question is how should such adaptation be defined and carried out.

- During a period of law-dispersion, agents operating under different versions of the law might attempt to exchange messages, without being aware that there is a difference between their respective laws. The question is how to handle the resulting inconsistencies. In particular, what should an agent $x/\mathcal{L}$ do when receiving a message sent under a previous version $\mathcal{L}^{-*}$ of the law? Such messages are called *ghosts*, as they are relics of past laws.

- Once a new law version $\mathcal{L}_c$ is introduced, how should one converge to the universal adoption of it by all agents of of the given system S, given that the ideal of universal and atomic adoption of the current law $\mathcal{L}_c$ is generally impossible.

It seems clear that these issues have no single answer, as they depend on both the system at hand and the law under which it operates. Therefore, we do not attempt to provide specific answers to these questions. What we provide, instead, are basic tools for resolving these issues, and a general approach for doing so in specific situations.

The rest of this paper is organized as follows. Section 2 describes the structure of evolving laws, and introduces the basic tools for supporting their evolution, illustrating them via an example. Section 3 describes the law-change process from a single agent perspective. Section 4 addresses the issue of evolution from a system-wide perspective. Section 5 discusses related work, and Section 6 concludes this paper.

2 The Structure of Evolving Laws

Laws in a lineage $\overline{\mathcal{L}}$ are identified via their name, as declared by the *Law($\mathcal{L}_i$)* clause in the preamble of each law; these names are constrained to be unique for a given lineage $\overline{\mathcal{L}}$. The text of each law $\mathcal{L}$ in the lineage is maintained at a URL, to be denoted by *url($\mathcal{L}$)*. Also, every law $\mathcal{L}$ in $\overline{\mathcal{L}}$, except the first law $\mathcal{L}_0$, must have the clause *previousLaw(url($\mathcal{L}^-$))* in its preamble, pointing to the law preceding it in the lineage. Finally, certificates are used to authenticate the laws in a lineage $\overline{\mathcal{L}}$ and to authorize a change when a new law is presented.

Evolving laws use several features introduced into LGI for this purpose. These include two additional events: *lawChanged*, and *ghost*, as well as a primitive operation: *changeLaw*. We will discuss these features in the context of the following example.

2.1 An Example

We will introduce here the first two versions $\mathcal{L}_0$ and $\mathcal{L}_1$ of an evolving law lineage $\overline{\mathcal{L}}$.

An Initial Law $\mathcal{L}_0$ of a Lineage: Let us consider the following, informally stated, capability-based access control policy. First, a subject `x` can access an object `o` if it holds a capability `cap(o)`. Second, the holder of a capability can delegate it *by lending*, for a specified period of time. That is, if `x` delegates its capability to `y`, then `x` looses this capability; `y` has the *obligation* to return it to `x` when the lending period has expired. Figure 2 displays an LGI law (written in pseudo code) that establishes this policy.

Rule $\mathcal{R}1$ of this law permits `x` to access `o` only when capability `cap(o)` is present in the control state (or simply state) of `x`, denoted by the clause $\exists$`cap(o)`. Rule $\mathcal{R}2$ enables the delegation by-lending of a capability, for a specified period of time. Note that one is allowed to delegate a given capability only if it has it in its control state; and if it is the original owner of that capability, i.e., if it does not have a pending obligation to return it. Also, the delegator loses the capability once it lends it. Rule $\mathcal{R}3$ shows how `y` acquires the delegated capability, along with an obligation to return it back to the delegator, after a prescribed period of time `t`.

Rule $\mathcal{R}4$ deals with the *exception* raised when the intended receiver of a delegated capability is not available. In such a case, the sender of the capability will re-acquire it in its state.

Rule $\mathcal{R}5$ is invoked when the obligation to return a lended capability comes due. The ruling for this event is to remove this capability from the state of its current holder, and to return it to the lender. Rule $\mathcal{R}6$ shows the re-acquiring of this capability at the initial owner.

```
Preamble: Law(ℒ0)

R1. upon sent(x,operation(Op),O)
        if ∃cap(o) do forward

R2. upon sent(x,delegate(cap(o),t), y)
        if ∃cap(o)&¬∃obligation(cap(O,X))
           do −cap(o),do forward

R3. upon arrived(x,delegate(cap(o),t), y)
        do +cap(o)
        do imposeObligation(cap(O,X),t)

R4. upon exception(x,delegate(cap(o),t),y)
        do +cap(o)

R5. upon obligationDue(cap(O,X))
        do −cap(o)
        do forward(Y,return(cap(o)),X)

R6. upon arrived(Y,return(cap(o)),X)
        do +cap(o)

R7.  upon arrived(x, pleaseUpdate(URL), y)
        do changeLaw(URL)
```

Figure 2. Law $\mathcal{L}_0$

Finally, Rule $\mathcal{R}7$ enables this law to be changed, as will be discussed in due course. (Note that this law does not show how the capabilities are acquired initially. LGI offers several ways to perform such initialization, which cannot be explained here due to lack of space.)

Law $\mathcal{L}_1$ of a Lineage: Suppose that after some time, the policy of the system needs to be changed such that: (a) delegation is permanent and not by lending; that is, the delegator does not loose the capability it delegates and no obligation is imposed on the delegatee; and (b) only *delegatable* capabilities, which have the form `cap(o,1)`, can be delegated, while capabilities of the form `cap(o,0)` cannot be delegated.

Law $\mathcal{L}_1$, displayed in Figure 3, establishes this policy via its first three rules. Rule $\mathcal{R}1$ allows `x` to access `o` when capability `cap(O,F)` is present in the state of `x`, regardless of whether F is `0` or `1`. Rule $\mathcal{R}2$ allows `x` to delegate its capability to `y` only if `x` has itself the capability, and the capability is delegatable (i.e. the delegability flag is `1`). Rule $\mathcal{R}3$ shows the acquiring of such copy-capability.

Rules $\mathcal{R}4$ through Rule $\mathcal{R}7$ are designed to support the transition from law $\mathcal{L}_0$ and law $\mathcal{L}_1$. They are discussed in the following section.

```
Preamble: Law(ℒ1)
          previousLaw(url(ℒ0))

R1. upon sent(x,operation(Op),O)
        if ∃cap(O,F) do forward

R2. upon sent(x,delegate(cap(O,F)), y)
        if ∃cap(O,1) do forward

R3. upon arrived(x,delegate(cap(O,F)), y)
        do +cap(O,F)


R4. upon lawChanged(ℒ0, ℒ1)
        if ∃cap(o)
           do −cap(o)
           do +cap(O,1)

R5. upon ghost(Event, ℒ0)
        if Event=obligationDue(cap(O,X))
        do fwd(Y,delegate(cap(O,1)),[X,ℒ0])


R6. upon ghost(Event, ℒ0)
        ifEvent=exc(Y,delegate(cap(O,t)),X)
           do +cap(O,1)

R7. upon ghost(Event, ℒ0)
        ifEvent=arrived(Y,return(cap(o)),X)
           do +cap(O,1)
```

Figure 3. Law $\mathcal{L}_1$

3 Single Agent Perspectives

This section focuses on a single agent operating under one of the law versions in $\overline{\mathcal{L}}$. We start with the basic mechanism for changing the law under which an agent operates. Then, in Section 3.2, we discuss the treatment of the "ghostly events" that an agent may confront due to its change of law. Finally, in Section 3.3 we discuss the handling of communication between agents operating under different law versions in $\overline{\mathcal{L}}$.

3.1 The Changing of a Law

A law change at an agent $x/\mathcal{L}$, from its law $\mathcal{L}$ to the next version $\mathcal{L}^+$, is a two stage process. First, while x operates under law $\mathcal{L}$ it invokes (via the ruling of its law) the primitive operation `changeLaw(U)`, where `U` is the URL of the new law $\mathcal{L}^+$. Second, the very next event in the life of x is the event `lawChanged(`$\mathcal{L}$`,`$\mathcal{L}^+$`)`, which occurs under its new law $\mathcal{L}^+$. The purpose of this event is to provide an opportunity for law $\mathcal{L}^+$ to complete the change of the law, in particular by making an appropriate update of its state. We now elaborate on these two stages of the law change; and will then point out that this change from $\mathcal{L}$ can be done to any law $\mathcal{L}^{+*}$ in $\overline{\mathcal{L}}$.

The First Stage of a Law Change: The circumstances under which the operation `changeLaw(U)` can be carried out by agent $x/\mathcal{L}$, depend on the law $\mathcal{L}$. Suppose, for instance, that x operates under our example law $\mathcal{L}_0$ displayed in Figure 2. According to Rule $\mathcal{R}7$ of this law, the operation `changeLaw(U)` will be invoked immediately upon the arrival of a message of type `pleaseUpdate( URL)`, presumably containing the URL of $\mathcal{L}_1$. So, the arrival of such a message is viewed by law $\mathcal{L}_0$ as a command to change the law (for simplicity, our example law does not specify how and by whom such a command can be sent.) Alternatively, once $x/\mathcal{L}$ is somehow notified of the existence of a new version of its law, the timing of the actual change might be left to the discretion of the actor of x, perhaps subject to some constraints, such as time limit.

Once the `changeLaw(U)` operation has been invoked, the changing process starts to take place. The first step is to load the new law $\mathcal{L}^+$ from the provided URL, and to check its certificate, if any, for validity. When loading the new law, the controller verifies that $\mathcal{L}^+$ belongs to $\mathcal{L}$'s lineage by checking the `previousLaw(`U`)` declaration in $\mathcal{L}^+$, as shown in the preamble of $\mathcal{L}_1$.

The Second Stage of a Law Change: Immediately after agent x replaces its law to $\mathcal{L}^+$ a `lawChanged(`$\mathcal{L}$`,`$\mathcal{L}^+$`)` event would be triggered at x, as the first event to take place after the law change. The arguments of this event represent the old law that has been updated, and the new one, which is thus the current law of agent x. The purpose of this event is to provide an opportunity for the new law to perform some initializations, such as: (a) adapt the control state of the agent x to the new law; or (b) perform some initialization operations, such as forwarding a message to some designated monitor notifying it of the law change.

For example, according to Rule $\mathcal{R}4$ in Figure 3, after x changes its law to $\mathcal{L}_1$, every term `cap(o)` in the state of x would be changed to a term of `cap(O,1)`, representing a delegatable capability for o under law $\mathcal{L}_1$.

Skipping Over Several Generations of $\overline{\mathcal{L}}$: So far we have discussed the changing of a law $\mathcal{L}$ to the immediately succeeding law $\mathcal{L}^+$. But the same operation `changeLaw(U)` is designed to change from $\mathcal{L}$ to any later version $\mathcal{L}$' of this lineage. Such skipping over generation of laws is done essentially as follows: (1) The controller of x verifies that $\mathcal{L}$' represents a subsequent version of $\mathcal{L}$. (2) The controller loads all the intermediary laws in the sequence $\langle\mathcal{L}, ..., \mathcal{L}'\rangle$ up to, and including, the final target law. (3) A sequence of `lawChanged` events is carried out for all the transitions in the law sequence; (4) Any event that might occur during the evaluation of the `lawChanged` sequence will become a ghost event scheduled under the new law $\mathcal{L}$' (see below about ghost events).

3.2 Self Ghosts

What we call "self ghost" are events that take place at an agent $x/\mathcal{L}^+$, but are caused by something in the history of x when it was operating under an earlier version of the law. The most important such ghosts have to do with obligations and exceptions. They are discussed below.

Exception Ghosts: Such ghosts can occur as follows. Suppose that a message sent by $x/\mathcal{L}$ fails, subsequently causing an exception event at x. But if x changes its law to $\mathcal{L}^+$ while either the message or the exception are in transit, then this exception will be triggered under the new law $\mathcal{L}^+$.

Rule $\mathcal{R}6$ in law $\mathcal{L}_1$ deals with such an exception ghost. If agent x delegates a capability under law $\mathcal{L}_0$, then immediately changes its law to $\mathcal{L}_1$, a subsequent exception ghost might occur at x. This rule will restore the capability whose delegation failed, but in the form `cap(O,1)` it needs to have under $\mathcal{L}_1$.

Obligations Ghost: Such ghosts can occur as follows: suppose that an agent $x/\mathcal{L}$ imposes an obligation on itself to come due after Δt seconds. If x updates its law to $\mathcal{L}^+$ before the Δt period has elapsed, the firing of this obligation will cause an obligation ghost event. Rule $\mathcal{R}5$ in law $\mathcal{L}_1$ is an example of dealing with such a ghost. Here, if an agent has a pending obligation to return a capability `cap(o)` to its lender, and the law has been changed to law $\mathcal{L}_1$, then the obligation is fired as a ghost event. As a result, the capability will be returned to the original owner, assuming (in this example) that the latter is still operating under the law $\mathcal{L}_0$. If the original owner already transitioned to law $\mathcal{L}_1$, then the message would be processed as a ghost, under Rule $\mathcal{R}7$, as discussed in the following section.

3.3 Communication Under Law Dispersion

We consider here a system in which different agents are operating under different versions of $\overline{\mathcal{L}}$, and individual agents do not generally know the version of the law under which other agents operate. Therefore, the sender of a message, say $x/\mathcal{L}$, has no choice but to assume that all other agents operate under law $\mathcal{L}$ as well. (This is consistent with the default of message sending under LGI).

The question discussed in this section is what should happen when this default assumption turns out to be incorrect, i.e., when the interlocutor of $x/\mathcal{L}$ operates on a dif-

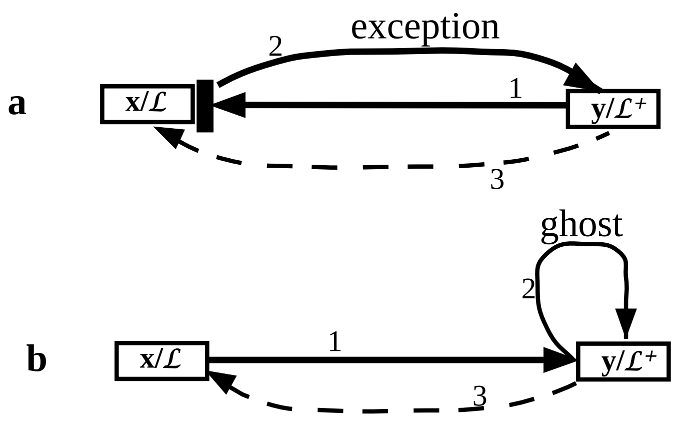

Figure 4. *Interaction Between Agents Operating Under Different Law Versions*

ferent version of the law. For simplicity we will examine this question by considering two agents $x/\mathcal{L}$ and $y/\mathcal{L}^+$ exchanging messages with each other. (The following discussion is valid for the case that several generations separate the law versions under which x and y operate.) We start with y sending a message to x, and then consider the other way around.

Messages from the Future: Suppose that $y/\mathcal{L}^+$ sends a message to $x/\mathcal{L}$, as depicted in part (a) of Figure 4. This message must be considered invalid by x, since it has been destined for an agent operating under $\mathcal{L}^+$, of which x knows nothing. Therefore, this message is blocked from being delivered to x; and y, the sender of this message, is notified with a specific exception event: `exception(x,M,y,destinationLawObsolete` `(`$\mathcal{L}$`))`, where $\mathcal{L}$ represents the law that x operates under. This exception is depicted by arrow number 2 in Figure 4.

One possible response of y to such an exception is to re-send the message after a certain period of time, expecting that y will update its law. Another type of response of y, now that it knows which law does x operate under, is to send a message to x explicitly stating its actual law. Such communication between agents operating under different, but known laws, is supported by LGI—it is called *interoperation*, or *export*. In particular, such an export—depicted by a dashed arrow in Figure 4—can be used to inform x that an update is available to its law. We will return in Section 4.2 to this type of response of y.

Messages from the Past: Suppose, now, that $x/\mathcal{L}$ sends a message to $y/\mathcal{L}^+$, as depicted in part (b) of Figure 4. When this message arrives at $y/\mathcal{L}^+$ it would trigger a *ghost event*, because y detects that the message has been sent under a law version that precedes the one y itself operates under.

What happens when a ghost event occurs depends on law $\mathcal{L}^+$ of y. For example, Rule $\mathcal{R}7$ in $\mathcal{L}_1$ handles the arrival of a ghost message at an agent. In this case, the ghost message represents the return of a delegated capability from an agent still operating under $\mathcal{L}_0$. Moreover, just as in the case of messages from the future discussed above, y can export a message to x (as depicted by a dashed arrow in Figure 4), informing it that an update is available to its law. Again, we return to this possibility in Section 4.2.

4 System Perspective

The issue to be addressed here is how does a system S, operating under an evolving lineage $\overline{\mathcal{L}}$, attempt to converge towards the universal adoption of its current law $\mathcal{L}_c$. We use the term "attempt," since, as has been pointed out earlier, complete convergence may not be achieved for long periods of time, if ever—in part, because the evolution of the law may be faster than its propagation, and because parts of the system may be temporarily disconnected from each other. In this section we discuss the process of convergence to a new current law $\mathcal{L}_c$ introduced by the system administrator.

Such convergence has two phases, which may overlap in time: (1) *seeding*, which results in having a subset of the agents in system S change their law to $\mathcal{L}_c$; and (2) peer to peer (P2P) convergence towards $\mathcal{L}_c$, which starts with the "seeds" planted in phase 1. We discuss these phases in the following two subsections. Finally, in Section 4.3, we describe a mechanism for universal and virtually atomic change to $\mathcal{L}_c$, which can be effective is some situations.

4.1 The Seeding of a New Current Law $\mathcal{L}_c$

This phase of convergence toward $\mathcal{L}_c$ has two parts: (a) dissemination of the new $\mathcal{L}_c$ to a subset S' of agents in S; and (b) the adoption of $\mathcal{L}_c$ by a subset S'' of S'.

The dissemination may be done in a variety of ways. In particular, the system manager may inform a subset S' of S of the existence of the new $\mathcal{L}_c$. (He may not be able to inform all agents in S because he might not know about all of them, or because some of them are unreachable, perhaps temporarily.) Another dissemination technique is by pull. That is, the various agents periodically query a given server about the existence of a new $\mathcal{L}_c$ for their system. The initiative for such periodic queries can come from the actor of an agent, or from its law, which imposes an obligation to do so periodically.

But informing an agent x about the new $\mathcal{L}_c$ does not necessary mean that x will immediately adopt this law. As we have explained earlier, an agents may decide to wait for a while before changing its law to $\mathcal{L}_c$, and it may even not do it at all. Therefore, the set S'' of agents that would end up being seeded with the new $\mathcal{L}_c$ may be smaller than the set S' of agents who were informed of it. But as long as S'' is not empty we can continue to the next phase of convergence.

4.2 P2P Convergence Towards $\mathcal{L}_c$

If the set S'' of agents seeded with law $\mathcal{L}_c$ is not the entire system, one needs to propagate this law to $S - S''$. This can be done incrementally, as a by-product of routine communication. That is, if an agent discovers during routine communication that its counterpart operates under an older version of the law, it should direct this agent to perform a law change.

More specifically, this P2P convergence, which needs to be supported by all versions in $\overline{\mathcal{L}}$, can be described in terms of the communication between two agents $x/\mathcal{L}$ and $y/\mathcal{L}^+$. As discussed in Section 3.3, when these two agents exchange messages under the assumption that they operate under the same law, the disparity between their laws will be detected at $y/\mathcal{L}^+$, and subsequently reported either as an exception, as depicted in Figure 4(a), or as a ghost message, as depicted in Figure 4(b). As also stated in Section 3.3, once y discovers the law $\mathcal{L}$ of x, y can send messages to $x/\mathcal{L}$ and not simply to x (which is interpreted as $x/\mathcal{L}^+$.) These messages, depicted in Figure 4 by dashed arrows, can be something like `pleaseUpdate(URL)`, where the URL is that of its own law $\mathcal{L}^+$. Such a message, intended to ask x to change its law to $\mathcal{L}^+$, would be effective if the law $\mathcal{L}$ of x is written to respond positively to this request. This can be done, in particular, as shown in Rule $\mathcal{R}7$ of the example law $\mathcal{L}_0$.

4.3 A Mechanism for a Virtually Atomic Law Change by the Entire System

This section is a summary of a technique—introduced in the first paper about LGI [8]—that attempts a universal and atomic change of law. This technique has two successive stages. The first stage uses a the two-phase commit protocol to get the agreement of all agents to carry out the law change. The second stage is divided into three successive parts: (a) a *relaxation* period, designed to allow agents to finish activities under the old law; (b) a *passive period*, designed to eliminate ghost messages, and inter-version message exchange; and (c) the law change itself, where the new law is set in place, by every agent, along with its updated control state.

The main limitation of this mechanism is that one cannot always get the agreement of all agents in the system to change the law. In particular, because a subset of these agents may be disconnected from the rest; and because the complete membership of the system, which may be dynamically changing, may not be known. But this atomic change can be used with only a partial participation in it, as an effective seeding stage (cf. Section 4.1) to be followed by the P2P convergence mechanism discussed in Section 4.2.

5 Related Work

Despite the importance of access control and high-availability requirements for many distributed systems, the problem of changing a distributed policy while the system continues to operate has received little attention so far. Perhaps the earliest attempt to address this issue has been published [8] in 1991 in the context of LGI; the present paper is an extension of that work, which is summarized in Section 4.3. The following are summaries of more recent attempts at this problem.

Ponder [5] is a mechanism for specifying management and security policies for a distributed system; its policies are enforced in a distributed manner. In [6], the authors describe a policy deployment and lifecycle model, including a policy change mechanism. This mechanism, similar to our previous model described in Section 4.3, requires that a policy is disabled with respect to all the agents in a policy set, then a new policy is deployed for that set. Although this model eliminates the potential of having ghost events and inconsistent communication between different versions of a policy, it can potentially produce large down times, where no policy is active in the system. Also, due to the lack of stateful character of Ponder policies, there is no mapping function between an old policy and the new policy.

DRAMA [4] is a policy-based network management system, designed to manage mobile ad-hoc networks. The policies are represented by event-condition-action rules concerned with configuration, monitoring, and reporting of management events in a network. DRAMA policies are enforced in a distributed manner by Policy Agents that are co-located with the managed network elements. Policy operations–such as enabling, disabling, or introducing new policies–are propagated between Policy Agents in a peer-to-peer manner, with some similarity to our convergence technique described in Section 4.2. DRAMA, however, is not concerned with controlling the communication between managed network elements, thus it does not have to address inconsistencies due to a partial update of a system.

XACML [7] uses a hybrid policy enforcement scheme, which involves distributed Policy Enforcement Points (PEP), co-located with protected resources, as well as a central Policy Decision Point (PDP), responsible for evaluating policies that apply to a given access request. Due to the centralized nature of the PDP, XACML is not confronted with the problems addressed in this paper, because its policy change takes place atomically with respect to the entire system.

The law-change model presented in this work has a number of affinities with the concept of automatically upgrading a distributed system. Most prominently, the model of Ajmani et al. [1, 2] shares a number of features, such as the peer-to-peer automatic discovery of inconsistencies between different versions of a software; a delayed, and controlled scheduling of an upgrade, as well as the mapping of the state as part of the upgrade process. There is a number of significant differences, however, between their model and our model. First, our model assumes that a law retains control over its own update procedure, thus defining and maintaining state consistency. Their model assumes an abstract, and unspecified consistent checkpoint for performing an update. Such a checkpoint can introduce severe limitations with respect to the types and the implementation of the objects that can be updated. Second, their model suffers from an unnecessary centralization for both seeding an update, and for coordinating the update schedule throughout the system. Our model does not rely on a centralized update database, and can use sophisticated controller-to-controller communication for both seeding the change and propagating it at individual components. Third, their model for dealing with inconsistencies between various version of a software uses the concept of a simulated object–an adapter that is responsible from dealing with, what we call, ghost communication events. Past simulator objects, however, have no access to the state of neither the older object nor the newer object, thus leading to un-handled inconsistencies. Last, our mechanism provides advanced access control to ensure that only proper changes take place, where a proper change can be defined according to multiple and flexible criteria. Their model relies on a centralized entity to decide unilaterally what a proper update is, without employing additional local information useful in such a decision.

6 Conclusion

We have introduced a model for *in vivo* evolution of the laws that govern a distributed system, when the governance (or access control) mechanism is strictly decentralized. We have argued that laws tend to be dispersed under such evolution, and that this dispersion, along with other aspects of *in vivo* evolution, can cause serious problems to the integrity of the system at hand.

We have described an approach for addressing these problems, thus providing for an orderly and safe process of evolution. We have implemented the necessary foundation that supports this approach under the LGI mechanism. This foundation for evolution is very lean, leaving most of the complexity of ensuring orderly evolution of the laws to the evolving laws themselves. Although this foundation has been implemented, it has not yet been included in the released version of LGI.

The main limitation of this paper is that it assumes that, in the absence of evolution, the entire system is governed by a single law. As we have seen, the evolution tends to disperse different versions of this law among the different system parts, but all these versions belong to a single lineage of laws. However, as we have shown in [3], a complex system may be governed by an ensemble of interoperating laws, generally organized into what we call a *conformance hierarchy*. The *in vivo* evolution of such an ensemble of laws is still an open question.

Another issue not sufficiently explored in this paper, is the structure of evolving laws. We have illustrated this structure in a single example in this paper. But a more comprehensive study of such structures would be useful.

References

[1] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 43–48, Lihue, Hawaii, May 2003.

[2] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[3] X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *LNCS 2808: Proc. European Symp. on Research in Computer Security (ESORICS)*, Oct. 2003.

[4] R. Chadha, H. Cheng, Y.-H. Cheng, J. Chiang, A. Ghetie, G. Levin, and H. Tanna. Policy-based mobile ad hoc network management. In *POLICY*, pages 35–44, 2004.

[5] N. Dulay, N. Damianou, E. Lupu, and M. Sloman. A policy language for the management of distributed agents. In *AOSE*, pages 84–100, 2001.

[6] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A policy deployment model for the ponder language. In *Proc. IEEE/IFIP International Symposium on Integrated Network Management (IM2001*, pages 14–18, 2001.

[7] S. Godic and T. Moses. Oasis extensible access control markup language (xacml), vesion 2.0. *http://www.oasis-open.org/committees/xacml/index.shtml*, March 2005.

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

[9] N. H. Minsky. *Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual)*, February 2006. (available at `http://www.moses.rutgers.edu/documentation/manual.pdf`