Enhancing the Dependability of Heterogeneous Distributed Systems by Governing their Message Flow

Naftaly H. Minsky

Rutgers University
Department of Computer Science
337 N. 5th ave. Edison NJ 08817 USA
Email: minsky@rutgers.edu

Abstract

This paper introduces an architecture of distributed systems that facilitates the implementation of dependable, mostly non-functional, *system properties*, i.e., properties that span an entire system, or a set of components dispersed throughout it. This architecture, called GDS, for *governed distribute system*, is based on a middleware that governs the flow of messages in a system, while being oblivious of the internals of the components that send and receive such messages. This middleware is stateful, thus sensitive to the history of interaction; and it is decentralized, and thus scalable.

The GDS architecture can help in establishing the overall structure of a system; and can have a significant impact on important non-functional qualities of it, such as the following: (a) establishing *regularities* over the system, rendering it more coherent, and easier to reason about; (b) providing for a degree of *trust* among the disparate actor of the system; and (c) enhancing the *security and fault tolerance* of a system. Although the impact of GDS would be strongest for heterogeneous and open systems, this architecture can be used effectively in distributed system in general.

Keywords: grid, middleware, dependability, governance, fault tolerance, architecture

Contents

1. Introduction

Heterogeneous distributed systems suffer from serious difficulties in establishing dependable *system properties*—i.e., properties that span an entire system, or a set of actors[1] dispersed throughout it. Such a property may, for example, be that all system components log certain kind of messages that they send or receive; or that all members of a group of collaborating agents comply with a given interaction protocol. And we consider, broadly speaking, a property P of a system S to be dependable if it satisfies the following conditions: (a) it is easy to verify that P is satisfied by S; (b) P is stable with respect to the evolution of the code of S—that is, it cannot be violated by most changes of this code; and (c) P is reasonably resilient to failures in S.

The difficulties in establishing dependable system properties is particularly serious in highly heterogeneous systems whose components may be written in different languages, may run on different platforms, and may be designed, constructed, and even maintained under different administrative domains. Such a distributed system is often said to be *open*[2] [5, 3]—because of a lack of effective constraints on the organization of the system as a whole, and on the internals of its disparate components. Distributed systems tend to become open when their size grows, and when they utilize components built by different vendors. Moreover, software *grids* are inherently heterogeneous, and partially open. Furthermore, systems are increasingly designed to be open, with the hope that this would make them more flexible. The concept of *service oriented architecture* [43] (SOA) is a prominent example of this trend, which is being adopted by a wide range of complex distributed systems, such as commercial enterprises, societal and governmental institutions, and various types of federated, or *virtual*, organizations, such as grids.

The following example illustrates the difficulty in establishing dependable system properties in an open distributed system. Consider an open system S that contains a distributed database, consisting a small number of database servers, which are being used by a large and heterogeneous set of clients. And suppose that each client can maintain locks over items on several database servers at a time. It is well known that such database usage would be *serializable*—an important criterion of correctness of concurrent transactions—if every client observes the *two-phase locking* (2PL) protocol [56], defined as follows.

[1] We use the term "actor" for an autonomous process of computation, that may be driven by software, by a physical device, or even by a person operating via some computation interface.

[2] The term "open system," as used here, has nothing to do with the concept of *open source*.

> *Once a client releases one of its locks, it does not acquire other locks until all its locks are released.* (This means that the interaction of a client with the distributed database is composed of two, perhaps repeating, phases: a *growing phase* of locking, and a *shrinking phase* of releasing all held locks.)

Now, suppose that one does not use a central *lock manager* for enforcing this protocol, because it would be a single point of failure and a potential bottleneck. The question, then, is, how can one ensure that all the disparate clients of the distributed database observe this protocol—rendering serializability of database access as system property.

The seemingly natural, approach for establishing system properties is *code based*. For example, to ensure that each potential client of a distributed database observes the 2PL protocol, each actor is to be programmed to comply with it. But doing so everywhere in an open system, is laborious, error prone, and hard to verify. And even if this protocol is established correctly in this manner, it would not be dependable because it can be violated by an inadvertent or malicious change in any component. Of course, one can try to avoid this difficulty by providing all potential clients with a stub designed to interact with the databases according to the 2PL protocol. But how can one be sure that some of the clients do not use this stab, and that they do not modify it to gain some advantage? Moreover, the use of such a stub is problematic when clients are programmed with different languages.

Note that one can do much better in a local—not distributed—system, via techniques like *reflection*, as under *meta object protocol* (MOP) [26]; or a governance technique like *aspect oriented programming* (AOP) [24], or *law-governed architecture* (LGA) [35]. These techniques can be employed—although less reliably—even for a distributed system if it is centrally managed , and if all its components are written in a single language for which such a technique are available. But none of these techniques is available for open distributed systems.

It has been suggested in [43], and elsewhere, that one can rely on very strong management and good software engineering practices to address these kind of problems. But while such, mostly human, disciplines are necessary in general, they cannot cope reliably with the complexities of a large open systems—due to lack of global knowledge of, and control over, the code of its component parts. *We need a more rigorous and more dependable solution for this serious problem.*

The Contribution of this Paper. We address this problem by adopting a level of abstraction that ignores the internal structure and behavior of the various actors of a system, focusing on the observable messaging-based interaction between them. Under this abstraction, we provide means for governing the flow of messages in the system, in a manner that is oblivious of the behavior of the actors that send and receive these messages. This is done, broadly speaking, by specifying how the flow of messages is to be controlled, and by enforcing this specification via an appropriate middleware—one that is stateful, thus sensitive to the history of interaction, and decentralized and thus scalable.

Of course, such governance of a system, which ignores the internals of its actors, can form mostly non-functional system properties. But such properties can help in establishing the overall structure of a system; and they can have a significant impact on important qualities of it, such as the following: (a) establishing *regularities* over the system, thus rendering it more coherent, and easier to reason about; (b) providing a degree of *trust* among the disparate actor of the system; (c) ensuring compliance with coordination protocols that are essential for distributed computing; and (d) enhancing the *security and fault tolerance* of a system. Such impact of the governance at this level of abstraction should not be surprising, if one considers an analogy with the governance of human societies—which ignores the thought and intention of people, dealing only with their interaction with each other, and with their environment.

This paper is a continuation of previous research that dealt with a fairly simple systems, which can be governed via a single monolithic protocol, such as the 2PL above. We have devised a middleware called LGI [34, 38] for specifying and enforcing such protocols. The protocol thus enforced on a system is called a *law*, and the system governed in such a law is called a *community*.

But a truly complex system, like an enterprise, a grid, or any kind of federated organization, cannot be governed effectively via a monolithic law. Its governance generally requires a whole collection of interlocking laws, which may be formulated by different stakeholders, in a semi-independent manner. We call a system governed by such a collection of laws, a *governed distributed system* (GDS). The difference between governing such a system and governing a single community is somewhat analogous to the difference between governing of a country, and governing some activity in it, like vehicular traffic. Therefore, the concept of GDS—which is the focus of this paper—is a major generalization of our concept of a governed community.

The Structure of this Paper. The rest of this paper is organized as follows. Section 2 is an outline of our previous work on the concept of governed distributed community, and of the LGI middleware devised for its support. Section 3 introduces the architecture of a GDS, and discusses various aspects of it, which includes a major extension of the LGI middleware. This section also provides an example of a GDS, which is an outline of an implemented case study of it. Section 4 discusses the potential impact of GDS on the dependability of distributed systems. Section 5 discusses related work. Section 6 discusses some open problems raised by the GDS architecture—and points out some inherent limitations of it. And Section 7 concludes this paper.

2. The Concept of Governed Distributed Community, and the Middleware that Supports it

The concept of governed distributed community (henceforth simply a *community*), and the LGI middleware that supports it, have been previously developed by the author and his students [34, 38]. This concept is outlined here in order to make this paper reasonably self-contained.

To appreciate the need for governed community, consider a distributed set of actors that need to interact with each other subject a given interaction protocol—such as the 2PL protocol introduced above—in order to collaborate effectively towards a common goal, or to compete safely over some resources. And suppose that the members of this set of actors cannot all be trusted to comply with this protocol, so that the protocol needs to be enforced. We call such an enforced protocol an *interactive law*, or simply a *law*, denoted generally by $\mathcal{L}$; and we call the set of actors operating subject to law $\mathcal{L}$ an $\mathcal{L}$-community. The formulation and enforcement of the interactive laws of such communities require an appropriate middleware, which arguably, needs to satisfy the following principles:

> *(P1) The middleware needs to support laws that are sensitive to the history of interaction between the actors of the community*, because coordination protocols, such as 2PL, are inherently so sensitive.
>
> *(P2) The enforcement of laws must be decentralized*, for several reasons: (a) for he sake of scalability; (b) to avoid having a single point of failure; and (c) to avoid the existence of a single target of attacks that can devastate the entire community.

As we have argued in [37], and briefly in Section 5, the *access control* mechanism—the convention means for controlling the flow of messages in a distributed systems—does not satisfy these two principles. We therefore designed, for our purpose, a new middleware called *law governed interaction* (LGI). The rest of this section focuses on this middleware, and on how it governs a distributed community.

2.1. The Law-Governed Interaction (LGI) Middleware—an Outline

We describe here a basic version of LGI that has been designed to support communities. A major enhancement of LGI developed for the support of GDS will be described in Section 3. This version of LGI has been published widely, and had been used, mostly experimentally, for a broad range of application—it has been released, along with its manual [36].

LGI has been designed for interaction between distributed actor via messaging, and it can handle various kinds of basic messaging types and protocols, specifically, TCP/IP, HTTP and SOAP, under both asynchronous, and synchronous messaging. And although most of our work has been conducted over Internet communication, there is a version of it that operates over ad hoc wireless networks, which has been used for several applications [13, 14]. Finally, we note that LGI makes no assumptions about the latency of messaging. We will start this outline of LGI by a description of its gist, and then discuss several aspects of it.

2.1.1. The Gist of LGI

Although the purpose of LGI is to govern the exchange of messages between the disparate member of a given $\mathcal{L}$-community according to its law $\mathcal{L}$, it does

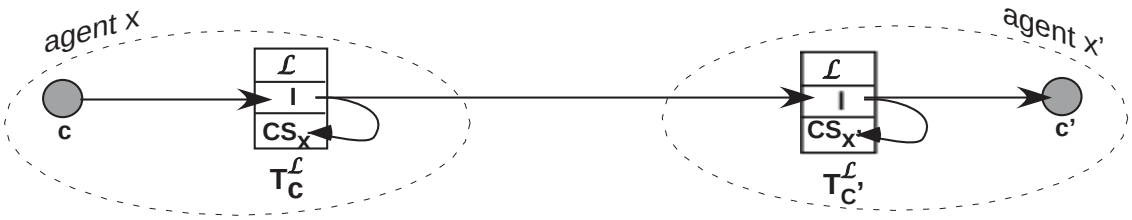

Figure 1: Interaction between a pair of LGI-agents, mediated by a pair of controllers under possibly different laws.

not do so directly. Rather, LGI controls the *interactive activities* of each actor c of an $\mathcal{L}$-community C, subject to the law $\mathcal{L}$. This control, over any given actor c, is strictly *local*, in that it is independent of the coincidental activities and state of any other actor in C.

This localization of control—which, as we shall see in Section 2.1.5 does not reduce the generality of LGI—is what enables its enforcement to be decentralized. Broadly speaking, law enforcement is done as follows: For an actor c to belong to an $\mathcal{L}$-community C is must interact with other members of C via a *private controller*, denoted by $T^{\mathcal{L}}_c$, that mediate the interactive activities of c, subject to law $\mathcal{L}$. $T^{\mathcal{L}}_c$ runs on a *generic controller* designed to interpret and enforce any well formed law.

We assume that there is a set of such generic controllers maintained by a trustworthy service called a *controller-service*, or CoS. The above mentioned controller, $T^{\mathcal{L}}_c$, is the one acquired by actor c form a CoS to mediate its interactive operations subject to law $\mathcal{L}$. The pair $\langle c, T^{\mathcal{L}}_c \rangle$ is called an $\mathcal{L}$-*agent*, which is said to be *animated* by the actor c. An $\mathcal{L}$-*agent*, or simply an agent, is denoted by symbols such as x or y. Figure 1 depicts the manner in which two such agents exchange a message. (An agent is depicted here by a dashed oval that includes an actor c and its controller, which generally reside away from its client c.) Note the *dual nature* of control exhibited here: the transfer of a message is first mediated by the sender's controller, and then by the controller of the receiver—both of which operating under the same law.

A key property of LGI-based communication is that a pair of $\mathcal{L}$-agents can recognize each other as such, and an $\mathcal{L}$-agent can communicate only with other $\mathcal{L}$-agents. And since laws are strictly enforced under LGI, it follows that the members of an $\mathcal{L}$-community can trust each other to observe the same law. (Note that this property will be generalized under GDS, which enables agents operating under different laws to interoperate, as we show in Section 3.3.4.)

2.1.2. On the nature of LGI's Laws

The function of a law $\mathcal{L}$ under LGI is to decide what should be done in response to the occurrence of certain events at any $\mathcal{L}$-agent x. This decision is called the *ruling* of the law for the event e that triggered it. More specifically, a law $\mathcal{L}$ that governs an agent x is defined in terms of three elements: (1) a set

E of *regulated events* (or, simply, "events") that may occur at any agent; (2) the *control-state* S_x associated with each agent x; and (3) a set O of *control operations*, that can be mandated by a law. To be more precise, all these elements are defined in the context of the controller of the agent, as the law is entirely oblivious of the internals of the actor c that animates this agent. We elaborate below on these three element of a law.

(1) The Set E of Regulated Events*:* This is the set of events that may occur at any agent x, and whose disposition is subject to the law under which x operates. E contains the following three types of events, among others: (a) the `arrived` event, which represents the arrival, at the controller of x, of a message intended for its actor; (b) the `sent` event, which represents the arrival, at the controller of x, of a message sent to anybody by its actor; and (c) the `adopted` event, which signals the moment of creation of the agent in question, by its actor adopting a controller with a given law.

(2) The Control-State *of an Agent:* The control-state (or simply "state,") S_x of an agent x, is maintained by its controller. This state is initially empty, but it can change dynamically in response to the various events that occur at x, subject to the law under which this agent operates.

(3) The Set O of Control Operations*:* These are the operations that can be included in the rulings of the law at any agent x. O contain, in particular, the following operations: (a) the `forward` operation, which forwards a message to another agent; (b) the `deliver` operation, which delivers a message to the actor of x—in effect, enabling this actor to access this message. The messages thus forwarded or delivered may be identical to those that the controller got from its actor or from another agent, respectively, but they can also be altered by the controller. (Note that the ruling of the law can also call for the state of x to be changed, but for notational reasons we do not include such instruction in the set O.)

We are now in a position to give a more formal definition of the concept of law, as follows.

Definition 1 (law). *Given the sets E and O, as defined above, and a set S of all possible control-states of an agent, a law $\mathcal{L}$ is a mapping:*

$$\mathcal{L} : E \times S_x \to S_x \times O^*. \tag{1}$$

In other words, for any given (e, S_x) pair, the law mandates a new state (which may imply an arbitrary state change), as well as a (possibly empty) sequence of control operations that are to be carried out in response to the occurrence of event e in the presence of state S_x.

Discussion. Several aspects of this concept of law are worth pointing out here.

(i) Note the interplay between the fixed law, and the dynamically changing state of a given agent. On one hand, the ruling of the law may depend on the current state of the agent, on the other hand the evolution of the state is

regulated by the law—although it is driven by the various event that occur at that agent.

(ii) *A law is local* at every agent x it governs, in that the ruling of the law is defined over the local events and state at x, and it can mandate only operations that can be carried out locally at x. Thus the law that governs an agent x can be enforced locally at x, without any access to other agents, and without any knowledge of the coincidental state of others.

(iii) A law cannot be inconsistent[3];and its ruling is never ambiguous. Indeed, by its definition, a law maps any given $(event, state)$ pair to a *single* ruling.

(iv) Note that this abstract definition of the concept of law does not specify a language for writing laws (which we call a law-language). This is for several reasons. First, despite the pragmatic importance of choosing an appropriate law-language, this choice has no impact on the semantics of LGI, as long as the chosen language is sufficiently powerful to specify all possible mappings of the form defined above. Second, specifying in this paper the details of of law-language would complicate it, without shading much light on the subject matter. In fact, the implemented LGI mechanism employs three different law-languages, which are based on: a)the logic-programming language Prolog; (b) Java; and (c) JavaScript. (We use the Prolog-based language for the our implementation of the 2PL protocol, in Section 2.1.4.)

2.1.3. On the Trustworthiness of Controllers, and on their performance

There are several reasons for trusting LGI-controllers, which we outline briefly as follows. First, controllers are generic and, designed to interpret and enforce any well-formed law. This enhances their trustworthiness, for the same reason that a well tested compiler tends to be more trustworthy, than the occasional program compiled by it. Second, the set of controllers used by a give community is to be maintained by a reputable organization, called controller service (CoS), whose business it is to provide correct controller to its customers. Such controllers would authenticity themselves by digital certificate signed by the CoS. Finally, using distributed LGI-controllers is more secure and more dependable than using a central reference-monitor—as under most access control mechanisms—because the distributed set of controllers does not constitute a single point of failure, nor a single point for attacking a community.

Regarding Performance, a comprehensive study of the overhead incurred by LGI control had been published in [38]. Broadly speaking, this overhead turns out to be relatively small, often smaller than the overhead incurred by control mechanisms such as XACML—beside being scalable. Moreover, this overhead is quite negligible for communication over WAN. The average contribution to this overhead by the computation in a controller was found—in circa 2000—to be around 50 microseconds. It is considerably lower with the present hardware.

[3]This does not mean, of course, that the a law cannot be incorrect, in the sense that it does not function as its writer intended.

2.1.4. Establishing the 2PL Protocol via Law $\mathcal{L}_{2PL}$

To illustrate the nature of of LGI-laws we display in Figure 2.1.4 a law $\mathcal{L}_{2PL}$ that establishes the two phase locking (2PL) protocol introduced in Section 1. This law is written in the Prolog-based law-language, which can be understood roughly as a set of even-condition-action (ECA) rules. The reader is not expected to be familiar with this law-language, or with Prolog. So, each of the rules of this law is followed by comments in italics. We will make here just the following introductory comment about this law and about the community it governs: Law $\mathcal{L}_{2PL}$ distinguishes between two kinds of members of the community governed by this law: (a) database servers, which authenticate themselves via a certificate (cf. Rule $\mathcal{R}1$) that identify them as having the role of a DB, thus obtaining the term `role(DB)` in their state; and all other actors that do not authenticate themselves and are assumed to be clients of the database. And note that a detailed explanation of a very similar version of this law is provided in [1].

2.1.5. On the Global Sway of the Local LGI Laws, and on their Expressive Power

The local nature of LGI laws raises the following two related questions: (a) is it possible to establish global properties of a system via the local LGI-laws; and (b) what is the expressive power of the local LGI laws. We will address both of these questions below.

Establishing Global Properties of a Community Via Local Laws. We distinguish between two kinds of global properties, which we call *regularities*, and *aggregate properties*. We define them below, and show how they can be established over a community.

(1) By regularity we mean here *compliance with a fixed principle*[4]. Formally, a regularity can be expressed as a universally quantified formula of the form $\forall x \in C\ P(x)$, where C is the $\mathcal{L}$-community in question, and $P(x)$ is a local property of agent x—that is, $P(x)$ is defined over the local state of x, and over local events at it. Regularities are the direct consequences of LGI, due to the fact that all members of an $\mathcal{L}$-community conform to the same law $\mathcal{L}$. An example of such regularity is the compliance with the 2PL protocol by all members of the $\mathcal{L}_{2PL}$-community.

(2) By an aggregate property we mean a property that depends on the state and behavior of all member of the community in question, or on a substantial part of its membership. As an example consider a law $\mathcal{L}$ that allows agents operating under it to issue purchase orders, but only as long as the total purchasing budget B of the community is not depleted. This property can be established via the local LGI law as follows.

Let one of the members of community C be an agent called `budget-office` (BO), whose state contains the current purchase budget of the community at large. And suppose that law $\mathcal{L}$ of C has two different modes of operations:

[4] See the American Heritage Dictionary

$\mathcal{P}$*reamble:*

authority(AdminCA, hashOfPublicKey).

this preamble enable the introduction of certificates signed, whose public key is identified by its hash (note that the initial state of every actor is empty).

$\mathcal{R}1$. `certified([issuer(AadminCA),subject(X),`
`attributes([role(DB)])) :-` **do**`(+role(DB))).`

an item role(DB) is added to the state of an actor, once its submits a certificate identifying itself as having a this role.

$\mathcal{R}2$. `sent(X,role(DB),_) :- do(+server).`

Under this law an agent can act a server—i.e., have a term `server` *in its initial control state—simply by sending a message* `role(DB)`*.*

$\mathcal{R}3$. `sent(C,lock(R),S) :- !shrinking@CS,`
`do(forward),`
`do(+lock(R,S,pending)).`

A message to lock resource `R` *is forwarded to its destination, only if the sender* `C` *is not in the shrinking phase. Also a term* `lock(R,S,pending)`*, denoting the pending status of this request, is added to the control state of* `C`*.*

$\mathcal{R}4$. `arrived(S,locked(R),C)`
`:- do(lock(R,S,pending)` $\leftarrow$ `lock(R,S,granted)),`
`do(deliver).`

If lock for resource `R` *is granted to an agent* `C` *then a term* `lock(R,S, pending)` *is replaced by* `lock(R,S,granted)` *to record that the lock has been acquired for* `R`*. The message then is delivered to the agent itself, in order to keep it informed.*

$\mathcal{R}5$. `sent(C, request(R,Param),S) :- lock(R,S,granted)@CS,`
`do(forward).`

A request by client `C` *regarding resource* `R` *is forwarded to server* `S` *only if the lock for* `R` *has been granted by* `S` *to* `C` *.*

$\mathcal{R}6$. `sent(C, unlock(R),S) :- lock(R,S,granted)@CS,`
`do(-lock(R,S,_)),`
`(!shrinking@CS` $\rightarrow$ `do(+shrinking);true),`
`(!lock(_,_,_)@CS` $\rightarrow$ `do(-shrinking);true),`
`do(forward).`

An unlock request is forwarded if the sender currently hold this lock; also, the correspondent `lock` *term is removed from the control state of the issuer. If this agent is not yet in its shrinking phase, it enters this phase by adding the term* `shrinking` *to its CS; and if* `R` *is the last locked resource held by* `C`*, the term* `shrinking` *is removed.*

$\mathcal{R}7$. `arrived(C,M,S) :- server@CS,`
`do(deliver).`

Any message that arrives at a server is delivered, without further ado.

$\mathcal{R}8$. `sent(S,locked(R),C) :- server@CS,`
`do(forward).`

`locked(R)` *messages sent by a server are forwarded without further ado.*

Figure 2: Law $\mathcal{L}_{2PL}$ that Ensures the Serializability of Transactions

First, when $\mathcal{L}$ resides in a controller associated with any member c of C except BO, it operates as follows: whenever c sends a purchase order to some vendor v the law causes this message to be rerouted to BO. Second, when $\mathcal{L}$ resides in a controller associated with BO, it operates as follows: (a) it forwards any purchase order it gets to its intended destination, *but only if* the global budget is sufficient for it; and (b) it updates the budget appropriately.

First note that law $\mathcal{L}$ described above is strictly local. It is true that implementing our aggregate property require some centralization of data, and some overhead. But this is involved just with purchase orders, which is expected to constitute a small minority of the total volume of interactions in a typical system. Such partial data centralization is not likely to degrade substantially the efficiency and the scalability of the community in question. There are, of course, other properties that require aggregation—such as the maintenance of unique naming, which we have implemented in [62]. But our experience with LGI suggests that the majority of global community properties required in practice, are of the regularity kind.

On the Expressive Power of the Local LGI Laws. We do not attempt here to define the expressive power of LGI directly. We will, instead, show that any policy that can be established via a centralized reference monitor (CRM) can be established via a local LGI Law. In other words, we will show that localization does not reduce the expressiveness of our control mechanism. We do this via the following simple construction.

Consider a community C, governed via a non-local law $\mathcal{L}$ enforced via a CRM, establishing a property P of C. We now show that P can be established via a local LGI law, as follows

Consider a community C' whose members are all the member of C, as well as an agent called G, which will serve as an aggregator. Now suppose that C' is governed by law $\mathcal{L}$' that operates as follows. First, when law $\mathcal{L}$' resides in an agent x of C, other than G, it operates in the following way: (a) whenever the actor c of agent x sends a message to some agent y of C the law causes this message to be rerouted to G instead; (b) when x gets a message from G, it delivers it to its actor c, without further ado. Second, when $\mathcal{L}$' resides in G, it operates precisely as law $\mathcal{L}$ operates when CRM got a message from some actor c—so it can establish the property P, just as it has been done by the CRM. Yet, law $\mathcal{L}$' is a strictly local law, because even when it resides in G it accesses only the state of G, which contains locally the entire aggregate state of the community, as it is in CRM.

Of course, this is just a theoretical construction, which shows that localization does not lose any expressive power. Although one may need some partial aggregations of state in practice, carried out via local laws, as we have shown above.

3. The Concept of Governed Distributed System (GDS)

Unlike a community, a complex distributed system S, such as a grid, cannot be governed effectively via a single monolithic law. Its governance generally requires a whole ensemble of semi-independent laws. For example, S may be partitioned into several semi-independent divisions, which may belong to different administrative domains, each of which may need to be governed by its own law, while conforming to the global law of the system at large. Moreover, a set of actors dispersed throughout system S, possibly crosscutting through several of its divisions, may have to interact with each other subject to a law that establishes a certain coordination protocol—such as our example law $\mathcal{L}_{2PL}$. And there may be many such communities in S that may need to operate under different communal laws. Some of these communities may need to interoperate, and their laws would all have to conform to the global system law. Furthermore, the various laws that thus participate in governing a given system may be defined and maintained by different stakeholders, with little or no coordination which each other—and the stakeholders themselves belong to different administrative domains.

We propose to cope with these complexities by organizing the set of of laws that collectively govern a given system into a *conformance hierarchy*, a device introduced into LGI, mostly for this purpose. We call a system governed by such a conformance-hierarchy of laws a *governed distributed system*, or a GDS—which is a far-reaching generalization of what we have called a community. And we refer to the hierarchical ensemble of laws that governs a GDS, as its *fabric*—denoting it by F—because it defines, in many ways, the underlying structure of this system[5]. The fabric of a GDS, organized in this way, turns out to be quite modular, as it simplifies the process of constructing a fabric, enhances its flexibility with respect of its evolution, and facilitates reasoning about the fabric itself, and about the system governed by it.

This section proceeds as follows: Section 3.1 outlines the concept of conformance hierarchy of laws; Section 3.2 defines of the architecture of GDS; Section 3.3 introduces various key aspects of GDS; Section 3.4 discusses a concrete and implemented example of a GDS; and Section 3.5 discusses several additional aspects of a GDS, which follow the example because they refer to it.

3.1. The Organization of Laws into a Conformance Hierarchy

LGI enables the organization of a collection of laws that collectively governs a single system, into what is called a *conformance hierarchy*. We denote here such a hierarchy of laws by the symbol F, because it is used in this paper to represent the *fabric* of a GDS. F is a tree of laws rooted by a law called $\mathcal{L}_R$, in which every law, except of $\mathcal{L}_R$ itself, conforms transitively to its superior law, in a sense to be described below. Moreover the conformance relation between laws

[5]We adopt here one of the definitions of "fabric" as an *underlying structure*—see the Webster Third New International Dictionary.

in F is inherent in the manner in which F is constructed, requiring no extra validation. For a formal definition of such hierarchy of laws, and a detailed example of its use, see [2]; here we provide just an informal introduction of this concept.

The Nature of Conformance of LGI-Laws. Several access control mechanisms [4, 19] defined conformance between policies basically as follows: *policy P' conforms to policy P if and only if P' is more restrictive than P, or equal to it.* But this would not do for LGI-laws, for several reasons, the most important of which is the following. The ruling of an LGI-law is not confined to a decision whether to approve or reject an action by an actor; it can also require some other actions to be carried out in response to an event, such as changing the sender's state in a specified manner, sending another message, or changing the message being sent and/or rerouting it. And it is generally not meaningful to ask if one such action is more or less restrictive than another. So, instead of using a uniform definition of conformance, based on restrictiveness, LGI lets each law define what it means for its subordinates to conform to it. This is done, broadly, as follows.

A law $\mathcal{L}$ that belongs to a conformance hierarchy F has two parts, called the *ground* part and the *meta* part. The ground part of a law $\mathcal{L}$ imposes constraints on the interactive behavior of the actors operating directly under this law—it has the structure defined by Formula 1. The meta part of $\mathcal{L}$ circumscribes the extent to which laws subordinate to it are allowed to deviate from its ground and meta parts. In particular, this allows a law, anywhere in this hierarchy, to make any of its provisions *irreversible* by any of its subordinate laws, by not permitting any deviation from it.

One application of such conformance is setting out defaults. For example, the root law $\mathcal{L}_R$ may prohibit all interaction between actors, while enabling subordinate laws to permit such interaction, perhaps under certain conditions. Alternatively, law $\mathcal{L}_R$ may permit all interaction, while enabling subordinate laws to prohibit selected interactions.

This very flexible concept of conformance is somewhat analogous to the manner in which the federal law of the US circumscribes the freedom of state laws to deviate from it. Such conformance turns out to be also useful for the governance of complex distributed systems, as we shall illustrate in Section 3.4.

The Formation of a Conformance Hierarchy of Laws. A conformance hierarchy F is formed incrementally via a recursive process described informally below. First one creates the root law $\mathcal{L}_R$ of F. Second, given a law $\mathcal{L}$ already in F, one defines a law $\mathcal{L}'$, subordinate to $\mathcal{L}$, by means of a law-like text called *delta*, denoted by $\Delta(\mathcal{L},\mathcal{L}')$, which specifies the intended differences between $\mathcal{L}'$ and $\mathcal{L}$. Now, law $\mathcal{L}'$ is derived from law $\mathcal{L}$ and $\Delta(\mathcal{L},\mathcal{L}')$, essentially *by dynamic consultation* during the interpretation of $\mathcal{L}'$, as described broadly below.

Consider the special case involving the root law $\mathcal{L}_R$, and its subordinate law $\mathcal{L}_s$ derived from $\mathcal{L}_R$ by the delta $\Delta(\mathcal{L}_R,\mathcal{L}_s)$. And consider an agent x operating under law $\mathcal{L}_s$. Now, when an event e occurs at an agent x it is first submitted

to law $\mathcal{L}_R$ for evaluation. Law $\mathcal{L}_R$ may consult the delta $\Delta(\mathcal{L}_R,\mathcal{L}_s)$ of $\mathcal{L}_s$ before deciding on its ruling—although it may also render its own ruling, not involving the delta. If consulted, the delta will do its own evaluation of this event, and will return its *advice* about the ruling to law $\mathcal{L}_R$. $\mathcal{L}_R$ would render its final ruling about how to respond to event e, taking the advice of the delta into account—but not necessarily accepting it, because this advice might contradict the meta part of $\mathcal{L}_R$. In this way, *the dynamically derived law $\mathcal{L}_s$ naturally conforms to its superior law $\mathcal{L}_R$, requiring no further verification.*

A notable property of this organization of laws is that interacting agents operating under laws in a common hierarchy can identify the position of each other's laws within this hierarchy. This can be done because every law carries the sequence of hashes of all the laws in the path from itself to the root $\mathcal{L}_R$.

3.2. The Architecture of a Governed Distributed System (GDS)

Formally, we define a GDS S as a four-tuple

$$\langle Ac, F, Ag, T\rangle,$$

where Ac is a set of *actors* that belong to S; F, called the *fabric* of S, is a conformance hierarchy of LGI-laws that collectively governs the flow of messages in S, while being oblivious of the internals of the communicating actors; Ag is a set of LGI-agents formed by actors in Ac operating under laws in F; and T, the trusted computing base (TCB [47]) of S, is a set of LGI-controllers that enforce the laws in F.

A GDS is portrayed schematically by Figure 3. The actors in Ac are depicted within the dotted rectangle by irregular shaded figures, representing their presumed heterogeneity, and the fact that the fabric F is oblivious to their internals. The controllers belonging to T are depicted by rectangles, inside the dotted oval that represents the CoS that maintains them. Finally, the dark irregular shapes on top of this figure depict actors, anywhere over the Internet, that do not operate under laws in F, and thus do not belong to the system in question; but F-agents may interact with them, if this is permitted by their laws. We now elaborate on the above mentioned elements of GDS, and on relationships between them.

(1) The set Ac of actors: This set consists of all the actors operating over the Internet that communicate subject to laws of the fabric F of S. But note that a given member c of Ac may not belong exclusively to Ac and thus to S, because we generally cannot rule out the possibility that c may also operate as part of another system, which may or may not be a GDS; and it may serve clients that have nothing to do with S. We cannot rule out this possibility because the fabric is oblivious of the internals of actors, which can be communicating in arbitrary manner. From the viewpoint of a given system S, such communication by its actors is called *foreign communication*, and the consequences of such communication to the concept of GDS are discussed in Section 3.3.6.

This architecture is silent on the membership of Ac, except that it is assumed to contain one distinguished actor with a specified role. It is called the *fabric*

server (or, F-server for short), and it role is to maintain the fabric F of this system.

(2) The fabric F: Although, as pointed up above, the laws in the fabric of a GDS are oblivious of the internals of the actors operating under them, the designers of these laws may not be so oblivious. A law designer may have some knowledge about the functionality of certain actors, or make various assumptions about them, and thus design his law accordingly. For example, $\mathcal{L}_{2PL}$ in Section 2.1.4 treats differently actors that serve as parts of a distributed database, and authenticate themselves as such. Also, a law can determine which actors may operate under it, in particular, by requiring actors to authenticate themselves in some way.

Note that due to the conformance nature of the hierarchical law ensemble F, its root law $\mathcal{L}_R$ has dominion over all the laws in it. This dominion is absolute for any provision of $\mathcal{L}_R$ defined as irreversible. Other provisions of $\mathcal{L}_R$ may be modified by subordinate laws, subject to constraints imposed by $\mathcal{L}_R$ on their modification. Consequently, this single law governs the entire fabric F, and thus, directly or indirectly, the entire system S.

(3) The set Ag of F-agents: An F-agent x is an actor c that communicates subject to some law $\mathcal{L}$ in F. More formally, an F-agents is a pair (cf. Section 2.1.1) $\langle c, T_c^{\mathcal{L}} \rangle$—namely, an actor c in Ac paired with the controller it adopted, under some law $\mathcal{L}$ in F. The messages sent by F-agents are called F-messages.)But if we want to be more specific about the law $\mathcal{L}$ under which an agent operate we call it an $\mathcal{L}$-agent and its messages we call $\mathcal{L}$-messages.)

Note that a single actor can animate several different F-agents, via different controllers, operating subject to the same or different laws. This may be the case, for example, when a single server provides several different services, possibly subject to different laws. Therefore, *it is the F-agents that are the loci of control by the fabric of GDS*, not the actors themselves, whose internals are not visible to F.

The structure of the fabric of a GDS is exemplified by Figure 4 that depicts the fabric used by the case study described in Section 3.4. This particular ensemble of laws is a three level conformance hierarchy, but it can, in general, be of any depth.

(4) The set T of LGI controllers: The controllers in T will be operating subject to various laws in F, this is unlike the case of an $\mathcal{L}$-community, all of whose controllers operate subject to a single law.

Note that assuming that the system S in question is large, it is perhaps best for the *controller service* (CoS) that maintains the set T, to be management under the administrative domain of this system itself, as part of its own *trusted computing base*.

And there are other ways for implementing trusted controllers. For example actors operating via their smart phones may have heir controllers built into their phones. We have done that, experimentally, in [13]. And there are ways for having such controllers reasonably well protected from attacks. Also, if one has a secure co-processor [54] attached to one's host, one can implement a controller on it.

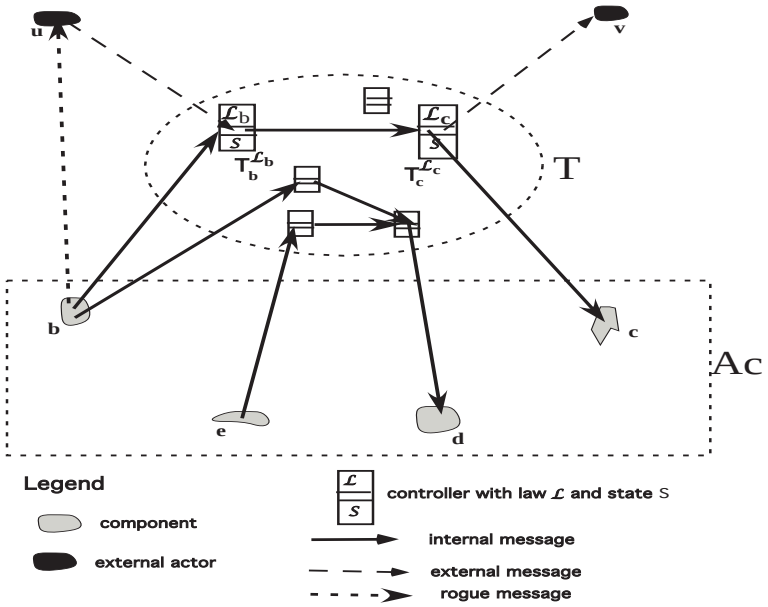

Figure 3: A Schematic Depiction of a GDS

3.3. Various Key Aspects of GDS

We start this section by discussing the consequences of having the entire fabric of a GDS maintained centrally, in F-server. We continue with the following aspects of the GDS architecture: (a) the reason for having laws of a GDS strictly enforced, and some of the implication of such enforcement; (b) the behavioral-trust modality, called *law-based trust*, induced by the GDS architecture; (c) the ability of F-agents operating under different laws to interoperate, which is based, in part, on law-based trust; (d) the freedom from inconsistencies between different laws of a fabric; and (e) the effect on a GDS of the existence foreign communication.

3.3.1. Why is the entire fabric of a GDS is maintained centrally, in F-server?

The main reason, besides convenience, for maintaining all the laws of the fabric F in the F-server—which is itself an actor of the GDS in question—is for enabling effective control over the evolution of F, as discussed in Section 3.5.3. But the reader may have the following questions about such a central placement of this essential part of a GDS: Will this centrality reduce the scalability of GDS? Will the F-server be a single point of failure of the GDS? And will it make a GDS more vulnerable to attacks? We addressed these questions briefly below.

First, the effect of maintaining F in a central place on the scalability of a GDS is negligible, because F needs to be accessed relatively rarely—mostly when F is constructed and modified, and when new F-agents are formed. Second,

F-service can be easily replicated—because it is not likely to change frequently—and thus would not be a single point of failure. Finally, the replicated F-service provides a good defense against attacks. Because if we have at least three replicas, then an attack on one of them can be discovered by frequent comparison of the hashes of the various replicas, replacing the compromised replica with a correct one.

3.3.2. About the Strict Enforcement of Laws

Laws are strictly enforced by the LGI middleware, rather then having violations of a law reported, with the expectation that a proper sanction for them would be applied at some time in the future—as it is done by OMNI [57], and by other multi-agent systems (cf. Section 5.1). This is because strict enforcement is important for dependability. For example, if the $\mathcal{L}_{2PL}$ is not enforced strictly, then the serializability property may be violated, and database clients may well enter into a deadlock.

However, it is not always possible to determine the legality or illegality of a single message, or the harm that a single message can cause. For this one may need to take into account past and future messages sent by possibly several agents. This situation requires suspicious messages to be reported, analyzed in the context of other information, and possibly respond to via an suitable sanction. This is precisely how OMNI enforces its laws, and how social laws are being generally enforced.

Such *report&sanction* approach to law enforcement can be facilitated under GDS in the following manner: Instead of blocking the potentially harmful message, or changing it in any way, the law can be written to log such a message in some logging service, which can later be analyzed by an appropriate manager or management software, which will determine the necessary sanction for it, if any—and then carry it out. And note that the logging of such potentially harmful message can be dependent on, because the law is strictly enforced. The analysis part of such *report&sanction* type of enforcement is beyond the scope of this paper.

3.3.3. Law-Based Trust: a Behavioral-Trust Modality Induced by GDS

There is, generally, little basis for trusting one's interlocutor over the Internet to behave in any particular manner—except for the relatively few actors with which one is familiar, or which have high level of reputation. This is largely the case also for the disparate actors that belong to an open distributed system. But a GDS provides for a general and useful mode of such a trust—not between the actors themselves, but between the F-agents animated by such actors. This mode of trust—called *law-based trust*, or *L-trust* for short—is defined as follows:

Definition 2 (law-based trust). *The actors of a pair of communicating F-agents can justifiably trust the observable—i.e., interactive—behavior of their interlocutor to comply with the laws under which it operates, even if these actors have no trust in each other.*

Moreover, each of these actors can identify the law under which its interlocutor operates, as well as the position of this law in the hierarchical structure of the fabric. Furthermore, each actor is able to obtain the text of the law of its interlocutor from the fabric-server.

This trust rests on the following assertions: (a) As part of their handshake, the controllers of the interacting agents exchange the certificates signed by the trusted CoS, which authenticate them as bona fide LGI controllers. (b) The controllers also exchange the sequence of hashes that identify the laws under which they operate, and the position of this law in the hierarchical structure of F. (c) The law enforcement by controllers is strict.

Finally, this mode of trust is fundamental to the GDS architecture as it facilitates some of its basic features such as interoperability, which is discussed in the following section. We will see other consequences of L-trust in due course.

3.3.4. Interoperation

We first discuss here how agents operating under different F-laws can interact with each other. We will then discuss interoperation between the F agents of a given GDS, and actors outside of this system. The technical details of interoperation, of both kinds, are spelled out in [36].

Interoperation Between F-Agents Operating Under Different Laws. To put this issue in context, we start by reviewing how interoperation is handled under conventional access control (AC). This issue has been has been addressed by several AC researchers, such as [20, 5, 29, 6, 61]. All of these share what we call the *composition* approach, which can be described as follows. Consider two parties operating under policies $P1$ and $P2$, respectively. To enable them to interoperate without violating their respective policies, one composes these policies into a single policy $P12$, which is consistent with both $P1$ and $P2$. The composition $P12$ is then to be fed into an appropriate reference monitor, which would mediate the interaction between the two parties. Unfortunately, composition of policies has several serious drawbacks (cf, [29]): (a) manual composition is laborious, and error prone; (b) automatic composition is computationally hard, and (c) composition is often impossible because $P1$ and $P2$ may actually be inconsistent. Moreover, as we have shown in [2], composition-based interoperation makes it very hard to establish multi-policy systems, and it renders changes of their policies extremely inflexible.

Under GDS, on the other hand, interoperation does not require composition of laws, and it is simpler and much more flexible. To see this, consider two F-agents $x1$ and $x2$, operating under laws $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Due to the dual mediation for every pairwise interaction via two different controllers (as shown in Figure 1), there is no need to compose $\mathcal{L}_1$ and $\mathcal{L}_2$ into a single law in order to enable $x1$ and $x2$ to interoperate. Rather, since each of these laws can recognize the other—due to L-trust—they may be written to enable interoperation, perhaps with certain restriction. They may, for example, allow the exchange of only certain types of messages which each other.

Moreover, since the two interacting laws belong to the same conformance hierarchy F, it follows that they both conform to their lowest common ancestor law in F; in particular, all the laws in F conform to the root law $\mathcal{L}_R$. If this commonality between $\mathcal{L}_1$ and $\mathcal{L}_2$ is sufficient for them to interoperate then they can do it seamlessly.

Interoperation Between an F-Agent of a Given GDS, and its Outside. The law under which an F-agent x operates can permit it to communicate with any actor z over the Internet. The law of x may impose some condition on such communication. In particular, it may require z to authenticate itself in a certain manner. Note that Figure 3 depict interaction between an F-agent b and an actor u that does not belong to the GDS in question.

3.3.5. Freedom from Inconsistencies Between the Laws of a Fabric:

Multi-policy systems regulated by access control mechanisms are plagued by the *policy inconsistency problem* [22, 19]. The conventional remedy to this problem is to apply various *conflict resolution techniques* for resolving such inconsistencies. But as has been shown in [10], these techniques are generally cumbersome, not very effective, and often lead to unexpected and undesirable results.

Fortunately, the fabric of a GDS is inherently free of inconsistencies. Indeed, a given law $\mathcal{L}$ in a hierarchy F cannot be inconsistent with its superior laws, because it is forced by its construction, to conform to them. And two different laws in F that do not reside on the same path from the root law, govern the interactive activities of different agents; so they cannot, by definition, be inconsistent. This inherent freedom from inconsistencies facilitates the construction and evolution of the fabric of a GDS, and makes it easier to reason about it.

3.3.6. Foreign Communication, and its Limited Effect on a GDS

While the fabric of a GDS has complete control over the interactive activities of its F-agents—i.e., of the messages sent and received by them—it does not, generally, control all the flow of messages in the system at hand. The reason for this is that except in some circumstances (discussed below) the actors, whose internals are not controlled by the fabric, can engage in "direct communication" (via TCP/IP, say), not subject to any law in F. In the context of a GDS, we call such communication *foreign*, because it is not bound by the fabric of this system. Figure 3 depicts foreign communication by the dotted arrow from actor b of S to the external actor u. And note that a fabric F may allow some F-agents to communicate with certain actors not operating under F—particularly with Internet sites that do not belong to the GDS in question. An example, depicted in Figure 3, is the interaction between an F-agent b, operating under law $\mathcal{L}_b$, with the same outside actor u. This messages is not foreign, as it is subject to a law in F.

Foreign communication by actors of a GDS is sometimes necessary, for example when an actor that operates as part of a GDS, also provides services to others over the Internet, in a manner not subject to any laws in F. But the

possible existence of foreign communication limits, somewhat, the control that a fabric has over a GDS. Suppose, for example, that F blocks all communication with a certain website w. This means that no F-agent can communicate with w. But the code of an actor that animates an F-agent can do so by direct messaging, and thus can reveal some information that should not be shared with w. This is, of course, a general problem, not specific to GDS or to LGI. For example the access control imposed by the reference monitor of the XACML mechanism [19] over the actors of an enterprise, does not really determine who can access whom, because actors can simply bypass this reference monitor.

However, as we point out at the top of Section 4, there is a wide range of system properties that can be established purely via the fabric of a GDS, regardless of any foreign communication that may exist in the system. And there is an even wider range of system properties that can be established by the fabric, under certain assumptions about the behavior of one, or very few, actors.

Nevertheless, sometimes one may want to eliminate foreign communication altogether. This is possible when the system in question is confined within an Intranet, or within a set of Intranets managed under a single administrative domain. Under these conditions one can force all actors in Ac—and the computers that host them—to communicate only as F-agents, by controlling the network, or networks, in which these host operate. For example, such control has been exercised via the firewalls attached to individual hosts, for the use of LGI to control the usage of distributed file systems [44]. A more systematic way for doing so should be possible under *SoftwareDefined Networking* (SDN) [31].

3.4. An Example of a GDS—Based on an Implemented Case Study

As a concrete view of a GDS, and particularly of its fabric, we provide here a simplified outline of a case study we have implemented. This case study simulated a federated enterprise called Acme consisting of two semi-independent divisions $D1$ and $D2$, maintained under different administrative domains, along with a *federation division* $D0$, maintained by the management of the federation at large. In particular, the distinguished F-server belongs to $D0$.

The fabric F of Acme is a three level conformance hierarchy of laws, depicted in Figure 4. The first level of this hierarchy—introduced in Section 3.4.1—consists of the root law $\mathcal{L}_R$. The second level contains the laws of the three divisions of the system, introduced in Section 3.4.2; as well as the crosscutting law $\mathcal{L}_{2PL}$ introduced in Section 2.1.4, as other crosscutting laws—the incorporation of such laws into F is discussed in Section 3.5.2. The third level, introduced in Section 3.4.3, is a collection of laws under which various individual actor may operate.

All these laws are described here informally, and they reflect only part of the laws used in our case study. Note that all but the root law are represented by their deltas[6], which specify the differences between the law in question and

[6]The concepts of *delta*, and of the *meta* part of a law (mentioned below) have been intro-

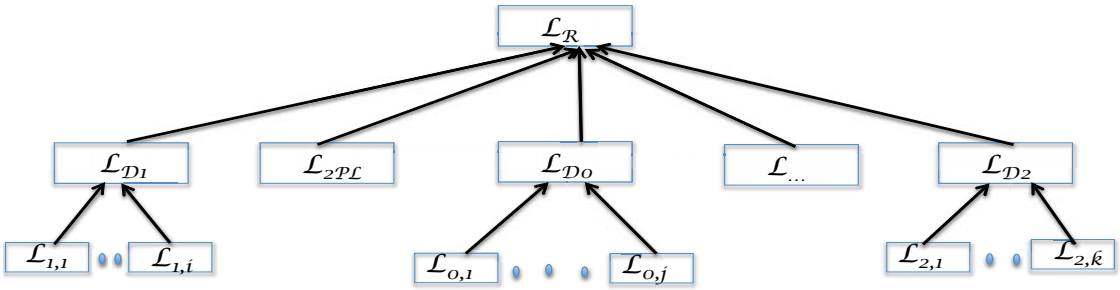

Figure 4: A Basic Hierarchical Law-Ensemble for the Acme System

its superior law. To see how such laws are actually written cf. [2] where a fairly sophisticated hierarchical ensemble of laws is introduced in detail. Here are some comments about our informal description of F.

We employ the following convention about the meta part of any given law $\mathcal{L}$ in the hierarchy: (a) if $\mathcal{L}$ has a rule that addresses a certain aspect of interactive activity of an actor subject to it—such as the sending of a certain type of messages—then this rule is *irreversible*, i.e., it cannot be deviated from by subordinate laws of $\mathcal{L}$, unless such deviation is explicitly permitted by the meta part of $\mathcal{L}$ (such permissions are denoted by bracketed texts in bold italics); and (b) if $\mathcal{L}$ is silent about certain aspect of interaction, then subordinate laws have the freedom of legislation about it. (This does not reflect the entire richness of the concept of conformance, but it will do for this example.)

3.4.1. The Root Law

As has already been pointed out, the root law $\mathcal{L}_R$ is the global law of the system, in the sense that all its provisions are shared by all the laws in F—modulo modification by subordinate laws, if such are permitted by $\mathcal{L}_R$. The main role of $\mathcal{L}_R$ is to establish broad system regularities and defaults. The following is the set of rules that thus govern Acme, which should be viewed as a small sample of rules that can be established by such a law. We elaborate on these rules, and motivate them, in the discussion that follows them.

1. **Authentication of actors:** To adopt an LGI-controller under this law, thus forming an F-agent, an actor c needs to authenticate itself via a certificate signed by $AcmeCA$, which we assume to identify the unique name of c, with respect to the Acme system, the role it plays, and the division to which it belongs; (note that this is a simplification, in practice actors may have several roles.) This authenticated identification of actors is stored in the state of their adopted controllers. ***[Subordinate laws may add conditions to this rule, and may require additional operations to***

duced in Section 3.1.

be carried out upon adoption, but they cannot weaken this rule.]

2. **Sender identification:** Every message sent is to be concatenated with the previously authenticated name, role, and division of its sender.
3. **Constraints over the interaction between F-agents:** The following two provisions are made by this rule: (a) all inter-division interactions are prohibited, ***[unless permitted by the corresponding subordinate division laws]***; and (b) all intra-division interactions are permitted, ***[unless prohibited by the subordinate division law in question]***;
4. **Establishing an Audit trail of inter-division interactions:** Every inter-division message would be logged in a specified logging service, upon its arrival.
5. **Providing operators with the power to control the system:** An F-agent that receives a message `stop(pattern)` sent by an operator—i.e., by an agent that has a role of an operator— would lose the ability to send or receive messages of the specified pattern; and if the pattern is "all", then it would lose the power to send all F-messages.

Discussion. The following is an elaboration on these rules, which provides some clarification and motivation for them.

Rule 1 provides Acme with a degree of control—exercised via its CA—over which actors can operate as F-agents. This provision is irreversible, governing all F-agents, although it can be tightened by subordinate laws, as it is by law $\mathcal{L}_{D1}$, below. Also, note that maintaining the certified identification of each actor in the state of its controller facilitates the enforcement of other rules of this law, such as rules 2 and 3.

Rule 2 provides the receiver of a message the ability to identify its sender. This is more informative and more trustworthy than identifying the sender by its IP address, which may not carry much meaning to the receiver, and which can be spoofed. Such identification can be useful in many ways. In particular it is used here for enforcing the constraints of Rule 3.

Rule 3 establishes two different types of access control provisions: Provision (a) prohibits all inter-division interaction, as a default, allowing subordinate division laws to permit any such interactions, as described in Section 3.4.2. (Note the unconventional nature of this type of conformance, where the subordinate division laws can be more permissive than their superior law.) Provision (b) is analogous to (a), with the opposite effect.

Rule 4 ensures dependable logging of all inter-division messages. Note that this rule can be stated here despite the fact that Rule 3 of the same law prohibits all inter-division messages—but if such an interaction would be permitted by the subordinate division laws, it would be subject to this rule.

Rule 5 enables system operators to prohibit any given F-agent from sending and receiving messages that fit a specified pattern. Such prohibition with the patter "all" would effectively remove the F-agent in question from a system by stopping all its communication. This is just an example of how one can endow some F-agents with a real power to control from a far what other F-agents

can do. (Note that this capability is analogous to the ability provided under SNMP [9] to operators to control network components, like computers, routers, etc. This rule is an example of how similar abilities can be provided at the application level, with respect to F-agents.

3.4.2. Division Laws

A division law, say law $\mathcal{L}_{D1}$ of division $D1$, is to be derived from the root law $\mathcal{L}_R$ via a delta $\Delta(\mathcal{L}_R,\mathcal{L}_{D1})$. One can reasonably assume that the writer of this delta has some idea of the intended structure of this division, and on the intended role and function of certain of its F-agents. This delta can, then, be used to impose this structure. For example, the delta of law $\mathcal{L}_{D1}$ makes the following three types of provisions.

(1) Constraint on the Composition of $D1$: Given that Rule 1 of law $\mathcal{L}_R$ permits its subordinate laws to add conditions on their adoption , this delta requires that actors adopting law $\mathcal{L}_{D1}$ would be authenticated as belonging to division $D1$.

(2) Imposing Constraints over Intra-Division Interaction: Recall that all intra-division interactions have been permitted by $\mathcal{L}_R$, as a default, allowing subordinates laws to impose arbitrary prohibitions on such interactions. So, this delta can impose any desired prohibition on the interactions between F-agents belonging to $D1$.

(3) Enabling Selected Inter-Division Interactions: Recall that inter-division interactions are prohibited by law $\mathcal{L}_R$, as a default, allowing subordinates laws to permit them. Note, however, that for an interaction between two divisions, say $D1$ and $D2$, to be enabled, it must be permitted by both $\mathcal{L}_{D1}$ and $\mathcal{L}_{D2}$—this is due to the local nature of our laws. For example, to permit a message from an F-agent $x1$ in $D1$ to an F-agent $x2$ in $D2$, law $\mathcal{L}_{D1}$ needs to permit $x1$ to send a message to $x2$, and $\mathcal{L}_{D2}$ needs to permit $x2$ to receive this message. Of course, such a permission may be formulated so it applies to whole sets of interaction types; for instance, the laws of the two divisions can have rules resulting in enabling certain types of messages to be exchanged between a certain sets of pairs of F-agents belonging to the two divisions.

3.4.3. Laws of Individual Agents

An actor c belonging to a certain division, may operate directly under the law of that division. But there are sometimes reasons for c to operate under its own law $\mathcal{L}_c$—subordinate to to its division. We discuss below two examples of such reasons.

First, as we shall see in Section 3.5.3, access to the F-server that maintains the fabric of the system need to be tightly regulated. This can be done by having F-server operate subject to a law $\mathcal{L}_{FS}$ built to carry out the required regulation.

Secondly, and more generally, suppose that an actor c is a web server, and that it makes certain promises to its clients about the services it provides. But such promises are not very credible if they are just stated, on the website of c say—particularly not in an open system, where the code of the service is not

known to its clients, and where this code can be changed without the client's knowledge. This inherent lack of trust is a serious and well known difficulty with services over the Internet. However, promises that can be formulated in terms of message exchange can be rendered trustworthy and dependable by formulating them as a law, and then providing one's services via a controller that enforces this law. The clients of such a service can trust these law-based promises due to the existence of L-trust, which has the following consequences: (a) the promise can be verified by studying the law—which is likely to be much smaller and simpler than the server's code; and (b) the law cannot be changed without the client's knowledge.

There are many examples of important promises that can be rendered trustworthy in this way, including such things as *money back guarantees*, the so called *service level agreements* (SLAs), *confidentiality*, etc. And, as pointed out before, a single actor may form different F-agents operating under laws that make different types of such promises. Below is a more detailed, but still informal, discussion of one type of such promises.

Server's Promise Made During a Conversation. The interaction between a server and its clients may involve a sequence of messages exchanged, according to some predefined protocol—such interaction is known as *conversation* [8]. During such a conversation, the server may make various promises to the client. For such promises to be dependable, they need to be enforced. For example, suppose that our server is a travel agent that provides for the following kind of conversation: A client x may request to reserve the right to buy a certain ticket at a particular price p, within a grace period t. If the server agrees, it should sell that ticket to x, if x pays for it within period t—which means that the server should not sell that ticket to anybody else within this time period. (Note that an LGI law that enforces such a promise, in a different context, has been described in [51].)

3.5. Additional Aspects of a GDS

We discuss in this section the following aspects of the GDS architecture: (a) the construction of a GDS; (b) the handling of crosscutting laws; and (c) the ability to regulate the evolution of the fabrics of a GDS.

3.5.1. The Construction of a GDS

We describe here the construction of a brand new GDS, which is quite straightforward. We do not address in this paper the more complex, and still largely open, issue of the complete conversion of a legacy system into a GDS. However, we do discuss here the grafting of some aspects of GDS into an otherwise conventional system.

The Construction of a GDS from Scratch. The construction of a new system S as a GDS can be described as consisting of two consecutive phases: (1) the design and construction of the *foundation* of S; and (2) the incremental construction of the rest of it. The *foundation* of S consists of two distinct parts:

(a) the root law $\mathcal{L}_R$ of the fabric F of S; and (b) a set of actors, which are required for the definition of $\mathcal{L}_R$. One of these is the F-server, which maintains the fabric of the system; and there may be other actors that may have to be included in the foundation.

Once the foundation of S is in place, the rest of it can be constructed *incrementally*, via steps of two kinds: (a) adding a law to F, subordinate to an existing law in it; and (b) having an actor adopt one of the existing laws in F, thus forming a new F-agent—we do not discuss here the programming of the actors themselves. These two types of steps can be carried out by different stakeholders in various orders, and perhaps concurrently. And recall that although each actor selects its own law, a law may impose constraints on the type of actors that may operate under it. In particular, in our example system Acme, law $\mathcal{L}_R$ requires an actor that attempt to adopt it to authenticate itself via a certificate signed by AcmeCA.

Note that the adoption of a given law in F is a voluntary act—it is not enforced by GDS architecture, nor is it enforceable. Yet, an actor may often be *virtually compelled* to operate under some law in F, or even under a specific such law—if it need some services provided only under this law. This is due to L-trust, which enables any F-agents to recognize the law under which its interlocutor operates, and thus decide whether or not to accept messages sent under a different law than its own. For example, if the databases of our example in Section 1 are written to accept only queries and updates sent under law $\mathcal{L}_{2PL}$, than anybody who wants to access this database would have to adopt this law.

Grafting Some Aspects of GDS into an Otherwise Conventional Distributed System. One may not want to subject an entire distributed system to GDS-type of governance; but just to incorporate some aspects of it into an otherwise conventional system. In particular, to carry out a group activity that requires compliance with a certain interaction-protocol, one may form a community governed by a law that enforces this protocol. For example, access to a distributed database can be done within a $\mathcal{L}_{2PL}$-community (cf. Section 2.1.4), ensuring serializability—even if the entire system does not operate as a GDS. As another example, issuing purchase orders in some enterprise can be done by members of a community, subject to a law that ensures compliance with some financial discipline, such as budgetary constraints, and auditing.

There may be many such governed communities in a given system, some of which may interoperate with each other. The las of these communities may even be organized into one or several different conformance hierarchies. All this without imposing a fully hierarchical fabric over the entire system. In fact such grafting may be an incremental way for converting an entire legacy system into a GDS.

3.5.2. The Handling of Crosscutting Laws

The hierarchical structure of the fabric of a GDS has many advantages, but it seems to rule out *crosscutting laws*, which are very important in many complex applications. To explain what we mean by "crosscutting laws," and the problem

they pose, consider a group G of actors that constitute a proper subset of all actors of Acme, and is dispersed throughout this system. For example, suppose that some members of G belong to division D1 of Acme, while other belong to division D2. Now suppose that members of G need to interact with each other subject to some law $\mathcal{L}_C$—we use C here for "crosscutting." A concrete example of such a law may be $\mathcal{L}_{2PL}$.

The problem is that there seems to be no place for defining this law as part of the hierarchical fabric F of Acme. It cannot be incorporated into the root law $\mathcal{L}_R$ of F, because $\mathcal{L}_C$ is not supposed to govern the entire system. It cannot be defined as subordinate to either law $\mathcal{L}_{D1}$ or law $\mathcal{L}_{D2}$ because group G crosscuts through both of them.

There is a simple resolution of this problem, which utilizes the ability of any actor to operate, simultaneously, under several different laws—provided that these laws agree to be adopted by the actor in question. So, law $\mathcal{L}_C$ can be defined as subordinate directly to the root law $\mathcal{L}_R$, just as we have placed law $\mathcal{L}_{2PL}$ in Figure 4.

It is worth pointing out, that this facility constitute an inherently distributed treatment of a class of *crosscutting concerns*. The issue of crosscutting concerns gave rise to the concept of *aspect oriented programming* (AOP) [25] for the non-distributed systems. The use of AOP in distributed systems is not possible, and certainly not dependable, for highly heterogeneous systems. The GDS architecture can support such concerns, if they involve only message passing.

3.5.3. Regulating the Evolution of the Fabrics of a GDS

Given the sway held by the fabric over the behavior of the system, it is critically important to be able to regulate the process of evolution of the fabric, in order to prevent careless and malicious changes to it.

In particular, it can be useful to control who can change which law of the fabric F, and under which circumstances should such changes be permitted. For example, one may want to require the consensus of several stakeholders for carrying certain changes. Moreover, if several changes of F are to be made by different stakeholders, perhaps concurrently, it may be necessary to establish a coordination protocol between them. Such regulation, we believe, can be useful despite the fact that the process of evolution is fundamentally unpredictable.

This types of regulation can be accomplished within the GDS architecture, because the fabric of a GDS is naturally *self regulatory*, in the following sense. As stated in Section 3.2, the fabric F of a given GDS is maintained in the F-server, which is itself an actor of the system in question. Therefore, since changes of F must be carried out by means of messages sent to F-server, and since these messages are governed by F, it follows that F can regulate its own evolution. This is a clean, and potentially very important, property of the GDS architecture.

4. On the Dependability of Governed Distributed Systems

The means provided by the GDS architecture for implementing dependable system properties is to establish them via the fabric of the system. We call such a property *fabric-based*, or an *F-property*; and we distinguish between a *perfect* F-property and an *approximate* one, as follows.

A perfect F-property is one that is established entirely by the fabric, making no assumptions about the behavior of any of the actor in Ac. For example the root law of our Acme case study (cf. Section 3.4.1) establishes several such properties. One of them, established by Rule 1, is that all its F-agents have been authenticated by a certificate signed by $AcmeCA$. Another example of a perfect F-property is that all the database clients that belong to the $\mathcal{L}_{2PL}$-community (cf. Section 2.1.4) comply with the 2PL protocol.

An approximate F-property is one established by F, *under the assumption* that one, or relatively few, actors satisfy a specified condition. An example of such a property is that the process of database use by the $\mathcal{L}_{2PL}$-community is serializable. For this property to be satisfied it is not sufficient for the clients of the database to operate according to the 2PL protocol—which is a perfect F-property, as pointed out above—one also need to make the following two assumptions about the few individual databases in question: (a) that they serve clients that operate subject to law $\mathcal{L}_{2PL}$; and (b) that they deal correctly with requests to lock and unlock database items.

We identify below three modes in which perfect F-properties are dependable.

1. *Ease of verification:* It is generally easier to verify a system property P if it is an F-property, than if P is implemented by the code of all or many system actors. This is because however large F itself may be, it is likely to be orders of magnitude smaller than the code of all system actors; and because of the lack of central control over the evolution of the code, in open systems.
2. *Independence* of the behavior of system actors: This makes such a property invariant under the evolution of the code of the system, and resilient to failures of it.
3. *Stability*, under the evolution of the fabric itself: While an F-property P can be changed by changing F, such changes can be regulated (cf, Section 3.5.3), so one would be able to depend on P not to change haphazardly during the evolutionary lifetime of the system.

These modes of dependability are closely satisfied even by approximate F-properties, if they depend on a small number of reasonably trustworthy actors. Therefore, henceforth we will often ignore the differences between these two types of system properties referring to both of them simply as F-properties.

There is wide range of mostly non-functional system properties that can be established via the fabric F of a system, any many of them can have beneficial effect on the system governed by F. In particular certain F-properties can make a system more coherent, easier to understand, more fault tolerant, and more secure—thus, altogether, more dependable. A sample of useful F-properties is outlined below.

4.1. A Samples of Useful Types of F-properties

Due to space limitation we limit ourselves here to *low laying fruit*, i.e., to types of F-properties that can be easily established under GDS—some of these are immediate consequences of this architecture. A much more comprehensive use of F-properties is described in a paper accepted for Journal publication [58], where the entire structure of a decentralized social network is established via F-properties. We proceed with the following three subsections that focus on trust, fault tolerance, and intrusion detection. Some of this is a review of previous discussion in this paper.

4.1.1. Engendering Trust

As pointed out in Section 3.3.3, there is, generally, little basis for the actors that belong to a given open system S to trust each other. Obviously, such lack of trust has an adverse effect on the dependability of such systems. But the fabric of a GDS can induce a degree of trust between its F-agents, although not between the actors that animate them. We comment below on two types of such trust, which we call: *behavioral trust, and identification trust*—both of which are due to the concept of L-trust introduced in Section 3.3.3.

The behavioral trust engendered by GDS is the trust in the interactive behavior of F-agent, to comply with the law under which it operates. For example, as shown in Section 3.4.3, L-trust enable an individual server make its promises, regarding its interactive behavior, trustworthy to all its clients. And the members of the $\mathcal{L}_{2PL}$-community can to trust each other to comply with the 2PL protocol.

An example of what we call an identification trust, is provided by Rule 1 and Rule 2 of the root law $\mathcal{L}_R$ of Acme (described in Section 3.4.1). These rules provides the receiver of any F-message an authenticated name of its sender, along with its role in the system in question, and with the division to which it belongs. This is more meaningful to the receiver of a message than identifying the sender by its IP-address or by its public key. Of course such certified identification can be done by other means. What is new here is the regularity of having the sender of every F-message being identified in this way.

4.1.2. Failure Prevention, Discovery, and Tolerance

We address here several different aspects of fault tolerance. And the techniques discussed here are applicable to both the conventional *system level* fault tolerance where the failing entity is a processor; as well as to the less well known, and in many ways harder, *application level* fault tolerance [16], where the failing entity may be a process, or, in our case, an F-agent.

Preventing Failures. Failures can sometimes be prevented by imposing a structure on a given system that rules out situations which may lead to certain types of failures. This is done by programming languages via strong typing and encapsulation, and can be done by the framework of an GDS in an analogous, but not similar, manner. For example, consider a group G of distributed actors that need to coordinate their activities, subject to a given protocol P. Such

coordination may fail by any member of G not following protocol P. Under GDS, however, if protocol P can be formulated in terms of message passing, then it should be possible to express and enforce it via a law $\mathcal{L}_P$ of the framework, which is to be employed by all members of group G for their coordination activity. We have done just that with our example 2PL protocol. And there are plenty of general conventional protocols, such as those devised to support *leader election*, *mutual exclusion*, etc., that can be handled in a analogous manner.

Facilitating the Detection of Failures, by Forced Heartbeat. We deal here with a situation where an actor—which may be a host, or some process running on one—halts prematurely. A well known way to detect such a failure of a given actor c, is to have c send heartbeat messages in a specified frequency to some monitor as long as it is alive. This would enable the monitor to conclude that c failed, if it did not receive heartbeat messages for a while. The problem here is that it may be hard to ensure that the lack of heartbeat messages is not due to c being buggy. And this is particularly hard to tell when dealing with a large set of of heterogeneous actors.

But the heartbeat discipline is easy to establish, for any number of F-agents, by having them operate subject to a law designed to send heartbeats in the required frequency, as long as the controller of an F-agent feels that its actor is alive, which it can do under LGI.)

Ensuring Fail-Stop Type of failures by F-agents. Fail-stop type of failures, traditionally defined for processors, is a failure where a processor halts, and never resumes its operations [49]. Such failures are easiest to tolerate. Indeed, as pointed out in [50], the well known *state machine* approach to fault tolerance is more efficient, and easier to carry out when a failure is fail-stop. And the failed processor can be safely replaced with an equivalent one, when one can be sure that the failed processor will not resume its operations. To facilitate fault tolerance, Schlichting and Schneider [48] devised a fairly elaborate axiomatic program verification technique for implementing processors that, with high probability, behave like fail-stop processors. But this technique has two limitation, from our viewpoint. First, it applies to processors, and not to application-level processes. And second, it depends on the code of the processor in question, and is, therefore, not applicable to open systems.

Fortunately, under GDS, we can easily force an apparent failure of any F-agent to be a fail-stop failure; simply by blocking its ability to communicate, once it is declared as a failed agent. This can be done in the following way, for example: Suppose that the root law $\mathcal{L}_R$ of the GDS in question contains a rule such as Rule 5 of the Acme system, described in Section 3.4.1. This rule empowers operators to block the ability of of any agent to communicate, and thus removing it, effectively from the system. Now anybody who decides that a given F-agent failed, can send a request to the an operator to remove it, in this sense, from the system.

It should be pointed out that a similar assurances of fail-stop failures can probably be provided under SNMP, for the processors managed by it—although

we do not know if this was ever done under SNMP. Anyway, SNMP cannot deal with the application-level concept of processes.

Replacement of a Failed Server. Consider a stateless service c of a conventional distributed system S, which failed, in a fail-stop manner. Suppose that there is an equivalent service c' which is believed to operate properly. Now, let our objective be to replace c with c', and to do so, with minimal loss, while the system continues to operate. The difficulty in achieving this objective is that the process of replacement takes time, however small, and any number of requests, sent by unknown clients, may arrives at the address of c during the replacement process, and thus may never be served or even acknowledged.

This difficulty can be addressed under GDS if c animates an F-agent x, and if it is c that failed, and not its controller[7]. This can be done broadly as follows: Suppose that the law of agent x, which we call $\mathcal{L}_r$ (r for "replacement") has the following, informally stated rules: (1) The controller t of x keeps continuously the list of all unanswered request. (2) t enters what we call *recovery mode* in response to two events: (a) if t gets the message `replacement started` from some properly authorized agent (say an operator); and (b) if t senses that its actor c failed, by either disconnecting from it, or by stopping to respond for sufficient amount of time. (3) During the recovery mode t keeps a list of request in its state, in the order of the arrival, and does not send them to its actors. This list will start with the list of unanswered requests it has. (4) Finally, the recovery mode ends when the controller gets the message `replacement-ends`, and it will then start sending the list of request it saved to the newly connected actor, in the order of their original arrivals.

Note, however, that this mode of recovery—which enables the client of the server c to be oblivious of the process of replacement of c with another server—can be accomplished in a traditional distributed system by having the interaction of c with its clients mediated by a surrogate that behaves according to law $\mathcal{L}_r$ described above. We can, however, accomplish more under a GDS. We can ensure that the same mode of recovery be available for a whole class of stateless services that have appropriate replacements.

4.1.3. Intrusion Detection and Prevention

The GDS architecture provides means for complementing the conventional anomaly based intrusion detection [55] (IDS), with a *specification-based* IDS that can detect intrusions on the fly, and can block some of them before they can cause any damage to the system. This is doable under GDS because an important part of the fabric is the specification of what are normative messages, with instructions to either block a message that deviates from the norm, or reports it for future disposition. Moreover, it is possible to remove the sender of an illegal message form the system (cf. Rule 5 in Section 3.4.1).

[7]Note that the the failure of the controllers of LGI can also be handled in a fault tolerant manner, which is very different than the mechanism discussed here, but this is beyond the scope of this paper.

Some of these measure are direct consequences of the GDS architecture, and do not require any extra cost. But it should be stressed that such specification-based IDS is not a replacement for the anomaly-based approach, but it is an effective complement for it, particularly for exposing and blocking *Trojan horses* that attack the system from inside.

It should also be pointed out the related work of Inverardi et al. [21] which, like GDS, monitors the flow of message in a system in a decentralized manner, blocking the non-normative ones. However, their mechanism has several serious disadvantages. Chief among them is that its specification of normative message flow, which is defined by a state machine, suffers from state explosion when the number of components grows. Consequently, this mechanism is unscalable in terms of its overhead and complexity—as was admitted by the authors of this paper—and is very inflexible with respect to changes of what is normative, which generally requires the construction of new state machine.

5. Related Work

We first discuss related work by other researchers; and then outline the relationships of this paper to previously published work by the author.

5.1. Related Work by Other Researchers

In a sense, this work is related to *reflection techniques* in non-distributed systems, such as *meta object protocol* (MOP) [26], *aspect oriented programming* (AOP) [24]. But here we focus only on mechanism to govern distributed systems. We will discuss several types of such mechanisms, more or less in the order of their similarity to this paper.

(1) The Concept of Architectural Model: This concept [17, 30] represents an external specification of certain aspect of a system—-mostly regarding interaction between system components—that are required to be implemented by its code. (Although this concept was originally proposed for central systems, it was widely adopted for distributed systems as well.) Since such a model is generally not enforced, it does not constitute a faithful description of the actual system, due to a gap that may exist between it and its architectural model, as argued in [40].

(2) Code-sensitive Governance of Distributed Systems: There is a host of governance mechanisms that have some dependency on the internals of the components of the system being governed. Some of them, like [46] and [7], require all system components to be written in a given language, and to employ a common programming discipline. Several papers, like [25], assume that all system components use *aspect oriented programming* in a coordinated manner. And papers that adopt the IBM approach to *autonomic system* [60], expect all system components to be *autonomic*. Obviously, these approaches to governance are not suitable for highly heterogeneous systems, which are the main target of this paper.

(3) Governance Based on Access Control (AC): There are many real applications and research papers, such as [4, 27, 28, 22, 45], that control the

flow of messages in a system, via some kind of access control (AC) mechanism, such as XACML [19] or RBAC [41]. This is done—just as under GDS—without any assumptions about the code of the interacting components. Some of these works identify their subject matter by phrases such as "policy based framework," "policy based systems," etc. But conventional access control has several inherent limitation—discussed in [37] and in [2]—which, by and large, makes it unable to support some of the key properties of the fabric of a GDS, namely: (i) sensitivity to the history of interaction (i.e,. being stateful); (ii) decentralization of enforcement; (iii) scalability; and (iv) modularity.

Yet, several, not entirely successful, attempts have been made for supporting some of these properties, by several AC mechanisms, as follows. *(i) Regarding statefulness:* We know of two attempts [23, 45] at making AC stateful, and thus sensitive to the history of interaction. But neither of them is scalable, as we have shown in [37], because they both use central reference monitor. *(ii) Regarding decentralization:* An important type of AC mechanism that is decentralized is the use of *distributed firewalls* for regulating the flow of messages in an enterprise [22]. Moreover, firewalls are sometimes partially stateful. However, while the role of firewalls in governing the flow of messages between a given system and the outside is important, firewalls do not control the exchange of messages between system components. *(iii) Regarding modularity:* An elegant AC mechanism called Oasis, designed by Moody and his group [4], features a concept of meta-policy, which is related to our conformance hierarchy—the basis for the modularity of our fabric. But Oasis differs from GDS in several ways, the most important of which is that the conformance in their hierarchy is not inherent to its structure, but needs to be verified, which generally needs to be done manually. A more recent design of a policy hierarchy [27] suffers from the same problem. Moreover, these hierarchies are not free of inconsistencies, and require various *conflict resolution techniques* for resolving them—as discussed in Section 3.3.5.

(4) Norms-based multi agent systems: Much of the literature on *multi-agents systems* (MAS) [3], and particularly on the type of MAS designed to support *electronic institutions* [12], recognizes the need for such systems to be governed by laws—which the called *norms* in this context. Many of these systems do not enforce their norms, but expect them to be observed voluntarily by their members, arguing that social systems—which constitute models for some of this work—do not, and cannot, enforce all their norms. However, some MAS projects, such as [57, 11, 12], do recognize the importance of enforcing norms. But they do so not by preventing violation of stated norms, but by detecting such violations after they occurred, and executing suitable sanctions for them. For example, OMNI [57], one of the most influential of these works, describe its approach to enforcement as follows: "In OMNI norm enforcement is not made in terms of direct control of a central authority over the internal goals or actions that the agents may take ... but through the detection of the violation [and by carrying out] the sanctions that are related to the violations."

Now, as we have pointed out in Section 3.3.2, there are indeed cases where this approach is appropriate. And we have explained there how it can be em-

ployed under GDS, whenever necessary. But in many, if not most, situation it is critical to prevent a violation whenever possible, because a violation of a law/norm may cause a serious damage to the system in question. Moreover, outright prevention of violation of norms is essential for the dependability of a system, because the effect of a violation may not be predictable, and because it is hard to depend on a sanction, which may or may not be carried out.

We are familiar with only one MAS project— AMELI [15]—that enforces its norms by preventing their violation. AMELI regulates the activity of its agents via a *governor*, in some analogy to the controllers of LGI—apparently, without knowledge of the much earlier publication about LGI. But unlike our controllers, the governors are stateless, and there is no concept of a set of such governors operating under a common norm, forming a community. AMELI has also a concept of a *scene*—somewhat analogous to our community—but the norm of a scene is enforced centrally via a *scene-manager*.

(5) Two additional related works: The works discussed here are conceptually the closest to this paper. First, our concept of fabric may be viewed as a concrete realization of the concept of *environments* of multi-agent systems, introduced by Weyns, Omicini, and Odell [59]. But while conceptually close to our approach, this paper does not present a specific architecture for governing a system.

Second, the work on law-governed systems, led by Carlos Lucena in Rio de Janeiro, Brazil [42] has many similarities to out to our work, particularly to our concept of governed community. Lucena's group even tackled the issue of fault tolerance in open distributed systems [18]. However, this project uses non-local laws, which are enforced mostly in decentralized manner, as compared to our local laws, with their decentralized enforcement.

5.2. Relationships to Previously published work by the Author

The contribution of this paper consists of the concept of GDS, and its impact on the dependability of distributed systems. In part, this work is based on some preliminary results published in several conference and invited papers, discussed in paragraph (1) below. Other aspects of this work, outlined in paragraph (2) below, are brand new. It should be pointed out that the concept of *distributed community* and the basic LGI middleware, are outlined in Section 2, only to make this paper self contained. These are not contributions of this paper.

(1) Published Conference Papers Related to the GDS Architecture, and to its impact: The most important of these is the paper [2] that defined the concept of *conformance hierarchy*. Then we have published initial and incomplete attempted at governing complex distributed system, in several application domains: coalition of organizations, like grids [2]; the exchange of medical record in US-wide health care services [33]; and monitoring of distributed systems [53]. We have also published three paper about fault tolerance—all in the simple context of a community. They are: (a) about preventing coordination failures [13]; (b) about self healing [32]; and (c) about reconfiguration [39].

(2) The novelties of this paper: First, the comprehensive architecture of GDS, provided by Section 3 is new, along with the following elements of it, which are critical for the formation of an effective GDS, and for its use: (a) the trust-modality called *L-trust*, between all actors of a GDS-based system; (b) the ability to control the evolution of a fabric of a GDS; (c) interoperation between agents operating under different laws; (d) a technique for imposing *crosscutting laws* on a system; and (e) the analysis of *foreign communication*, and of its effect on GDS.

Second, this paper provided a broad outline of the impact of GDS on the dependability of distributed systems, which has not been published so far.

6. Open Problems Raised by the GDS Architecture—and its Inherent Limitations

This is a work in progress, because the concept of GDS raises some issue that are left open by this paper, some of which are discussed in Section 6.1. And any system architecture has some cons along with its pros. Some of the inherent drawbacks of GDS are discussed in Section 6.2

6.1. Some Open Problems

The GDS architecture, as presented in this paper, raises some open issues that need to be addressed for this architecture to attain its full potential. Three of these issues are outlined below:

(1) Validation of the Practicality of the GDS Architecture: The experimental case study of a GDS we have implemented, and outlined in Section 3.4, constitutes a proof of concept of GDS. But the validation of the practicality of this architecture requires experimental work on a real, or at least, realistic, distributed system. Such an experiment is yet to be carried out.

(2) Converting a Legacy System into a GDS: Although the construction of a GDS from scratch, as described in Section 3.5.1 is straightforward, the conversion of a legacy system into a GDS is much more complex—and it would facilitate wide adoption of this architecture by the industry. We have began working on this problem, but more work is necessary.

(3) Issues concerning the evolution of the fabric of a GDS: We have already provided for the *control* of the evolution of the fabric of a GDS, as discussed in Section 3.5.3. But there are additional issues involved with such evolution, which have not been resolved yet. These include the following, in particular: (a) how to carry out a change in a law $\mathcal{L}$ of F, given the dependencies on it by a law $\mathcal{L}$' subordinate to $\mathcal{L}$, or interoperating with it; and (b) how to change the fabric of a GDS while the system continues to operate. We have solved this problem for a community governed by a single law [52], but doing so for a multi-law fabric is much more challenging.

6.2. Inherent Drawbacks of the GDS Architecture

The following are two drawbacks of the GDS architecture, the first is self evident, and the second more subtle, and potentially more serious.

(1) Enforcement Overhead: Obviously, subjecting all communication between F-agent to LGI-based mediation for the enforcement of the laws of F involves an overhead. We have argued in Section 2.1.3 that such overhead is comparable, and much more scalable, than that of using other popular middleware, like access control. Yet, such overhead may be unacceptable to some systems.

(2) The power invested in the fabric over its GDS: The fabric has an enormous power over the behavior of the system S governed by it. It is this power that allows us to implement system properties of S dependently. But such power has a dark side: mistakes made in the laws of the fabric of S can have a devastating effect on its behavior. This risk can be tamed by controlling the evolution of the fabric of a GDS, but it cannot be entirely eliminated.

So, we have here a conflict between the benefits of governing a system by its fabric, and the resulting risks. One needs to be aware of this conflict before deciding to employ the GDS architecture. But it should be pointed out that an analogous conflict exists in other, well know and much used, attempts at governing a system from above, as it where. These include, *access control*; as well as—for non distributed systems—*reflection*, like under *meta object protocol* (MOP) [26], and *aspect oriented programming* (AOP) [24].

7. Conclusion

This paper introduces an architecture of distributed systems that facilitates the implementation of dependable, mostly non-functional, *system properties*, i.e., properties that span an entire system, or a set of components dispersed throughout it. This architecture, called GDS, for *governed distribute system*, is based on a middleware that governs the flow of messages in a system, while being oblivious of the internals of the components that send and receive such messages. This middleware is stateful, thus sensitive to the history of interaction; and it is decentralized, and thus scalable.

The GDS architecture can help in establishing the overall structure of a system; and can have a significant impact on important non-functional qualities of it, such as the following: (a) establishing *regularities* over the system, rendering it more coherent, and easier to reason about; (b) providing for a degree of *trust* among the disparate actor of the system; (c) ensuring compliance with coordination protocols that are essential for distributed computing; and (d) enhancing the *security and fault tolerance* of a system. Although the impact of GDS would be strongest for heterogeneous and open systems, this architecture can be used effectively in distributed system in general.

It should be pointed out, that this is a *work in progress*, in two respects. First, although the implemented case study of GDS, described in Section 3.4, constitutes a *proof of concept* of this architecture, the real usefulness and effectiveness of this architecture needs to be validated by applying it to one or more

real (or realistic) large scale and complex distributed systems. Such validation is yet to be done. Second, as stated in Section 6, the GDS architecture introduced in this paper raises some open issues that need to be addresses, for GDS to attain its full potential.

References

[1] X. Ao, N. Minsky, T. Nguyen, and V. Ungureanu. Law-governed communities over the internet. In *Proc. of Fourth International Conference on Coordination Models and Languages; Limassol, Cyprus; LNCS 1906*, pages 133–147, September 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[2] Xuhui Ao and Naftaly H Minsky. Flexible regulation of distributed coalitions. In *Computer Security–ESORICS 2003*, pages 39–60. Springer, unknown, 2003.

[3] A. Artikis, M. Sergot, and J. Pitt. Specifying norm-governed computational societies. Technical report, Department of Computing, Imperial College of Science Technology and Medicine, London, 2006.

[4] A. Belokosztolszki and K. Moody. Meta-policies for distributed role-based access control systems. In *Proc. of the IEEE 3rd International Workshop on Policies, Monterey, California*, pages 106–15, June 2002.

[5] C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 51–66, September 1998.

[6] P. Bonatti, S. D. Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 164–173, 2000.

[7] Mauro Caporuscio, Antinisca Di Marco, and Paola Inverardi. Model-based system reconfiguration for dynamic performance management. *Journal of Systems and Software*, 80(4):455–473, 2007.

[8] F. Casati, E. Shan, U. Dayal, and M. Shan. Business-oriented management of Web Services. *Communications of the ACM*, 46(10):55–60, Oct. 2003.

[9] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP), 1990. RFC: 1157, available from `http://www.ietf.org/rfc/rfc1157.txt`.

[10] Ritu Chadha. A cautionary note about policy conflict resolution. In *Military Communications Conference, 2006. MILCOM 2006. IEEE*, pages 1–8. IEEE, 2006.

[11] Natalia Criado, Estefania Argente, Antonio Garrido, Juan A Gimeno, Francesc Igual, Vicente Botti, Pablo Noriega, and Adriana Giret. Norm enforceability in electronic institutions? In *Coordination, organizations, institutions, and norms in agent systems VI*, pages 250–267. Springer, 2011.

[12] Marcos de Oliveira, Enyo Golçalves, and Martin Purvis. Institutional environments: A framework for the development of open multiagent systems. In *Advances in Artificial Intelligence–IBERAMIA 2014*, pages 560–571. Springer, 2014.

[13] Rishabh Dudheria, Wade Trappe, and Naftaly Minsky. Coordination and control in mobile ubiquitous computing applications using law governed interaction. In *Proc. of the Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM) Florence, Italy*, pages 247–256, October 2010.

[14] Rishabh Dudheria, Wade Trappe, and Naftaly Minsky. Law governed peer-to-peer secondary spectrum marketplaces. In *8th International Conference on Cognitive Radio Oriented Wireless Networks (CROWNCOM 2013)*, July 2013.

[15] Marc Esteva, Bruno Rosell, Juan A Rodriguez-Aguilar, and Josep Ll Arcos. Ameli: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 236–243. IEEE Computer Society, 2004.

[16] Vincenzo De Florio and Chris Blondia. A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys (CSUR)*, 40(2):6, 2008.

[17] D. Garlan. Research direction in software architecture. *ACM Computing Surveys*, 27(2):257–261, 1995.

[18] A.C. Gatti, C.J.P Lucena, and J.P. Brio. On fault tolerance in law-governed multi-agent systems. In *Proc. of ICSE'05, 4th Int. Workshop on Soft. Eng. for Large-Scale Multi-Agent Systems*. ACM, July 2006.

[19] S. Godic and T. Moses. Oasis extensible access control. markup language (xacml), version 2. Technical report, Oasis, March 2005.

[20] L. Gong and X. Qian. Computational issues in secure interoperation. *IEEE Transctions on Software Engineering*, pages 43–52, January 1996.

[21] Paola Inverardi and Leonardo Mostarda. A distributed intrusion detection approach for secure software architecture. In *Software Architecture*, pages 168–184. Springer, 2005.

[22] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[23] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and VS Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

[25] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 49–58, September-October 2005.

[26] G. Kiczales, J.D. Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[27] D. Lee, S. Ahn, and M. Kim. A study on hierarchical policy model for managing heterogeneous security systems. In *Proceedings of the 2011 international conference on Computational science and its applications, ICCAS11*, June 2011.

[28] B. Li, L. Zhao, J. Zhu, and J. Wu. A policy-based adaptive web services security framework. *Journal of Software*, 6(12), December 2011.

[29] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *Proc. of the IEEE Symp on Security and Privacy*, May 2002.

[30] Nenad Medvidovic, David S Rosenblum, and Richard N Taylor. A language and environment for architecture-based software development and evolution. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 44–53. IEEE, 1999.

[31] Marc Mendonca, Katia Obraczka, and Thierry Turletti. The case for software-defined networking in heterogeneous networked environments. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 59–60. ACM, 2012.

[32] N. H. Minsky. On conditions for self-healing in distributed software systems. In *In the Proceedings of the International Autonomic Computing Workshop Seattle Washington*, June 2003. (available at `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[33] Naftaly Minsky. A model for the governance of federated healthcare information systems. In *Proc. of the IEEE Internationaal Symposium on Policies for Distributed Systems and Networks, George Mason University, USA*, July 2010.

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

[35] Naftaly H. Minsky. Law-governed systems. *The IEE Software Engineering Journal*, September 1991.

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

[37] Naftaly H Minsky. Decentralized governance of distributed systems via interaction control. In *Logic Programs, Norms and Action*, pages 374–400. Springer, unknown, September 2012.

[38] Naftaly H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[39] Naftaly H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *Proc. of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, March 1996. (available from `http://www.cs.rutgers.edu/~minsky/`).

[40] G.C. Murphy, D. Notkin, and K. Sullivan. Software reflection models: Bridging the gap between source and high level models. In *Proceedings of the Third ACM Symposium on the Foundation of Software Engineering*, 1995.

[41] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000.

[42] Rodrigo Paes, Gustavo Carvalho, Maíra Gatti, Carlos Lucena, Jean-Pierre Briot, and Ricardo Choren. Enhancing the environment with a law-governed service for monitoring and enforcing behavior in open multi-agent systems. In *Environments for Multi-Agent Systems III*, pages 221–238. Springer, 2007.

[43] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Kramer. Service-oriented computing: A research roadmap. In Francisco Cubera, editor, *Service Oriented Computing (SOC)*, number 05462 in Dagstuhl Seminar Proceedings. Internationales Begegnungs, 2006.

[44] Zhijun Phan, Tuan He and Thu D. Nguyen. Policies over standard client-server interactions. *JOURNAL OF COMPUTERS*, 1(1), April 2006.

[45] C. Ribeiro and P. Ferreira. A policy-oriented language for expressing security specifications. *International Journal of Network Security*, 5(3), November 2007.

[46] Jonathan C Rowanhill, Philip E Varner, and John C Knight. Efficient hierarchic management for reconfiguration of networked information systems. In *Dependable Systems and Networks, 2004 International Conference on*, pages 517–526. IEEE, 2004.

[47] John M Rushby. *Design and verification of secure systems*, volume 15. ACM, 1981.

[48] Richard D Schlichting and Fred B Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[49] Fred B Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984.

[50] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[51] C Serban, Y. Chen, W. Zhang, and N. Minsky. The concept of decentralized and secure electronic marketplace. *The Journal of Electronic Commerce Research (JECR)*, 8(1-2):79–101, June 2008.

[52] C. Serban and Naftaly Minsky. In vivo evolution of policies that govern a distributed system. In *Proc. of the IEEE International Symposium on Policies for Distributed Systems and Networks, London*, July 2009.

[53] C Serban, W. Zhang, and N. Minsky. A decentralized monitoring mechanism for distributed applications. Technical report, Rutgers University, March 2008.

[54] S.W. Smith and V. Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *3rd USENIX Workshop on Electronic Commerce*, August 1998.

[55] Matthew Stillerman, Carla Marceau, and Maureen Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42(7):62–69, 1999.

[56] A. Tanenbaum, R. Van Renesse, H. Staveren, G.J. Sharp, S.J. Mullender, and G. Rossum. Experiences with the amoeba distributed system. *Communications of the ACM*, December 1990.

[57] Javier Vázquez-Salceda, Virginia Dignum, and Frank Dignum. Organizing multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 11(3):307–360, 2005.

[58] Zhe Wang and Naftaly Minsky. A novel, privacy preserving, architecture for online social networks. *EAI Endorsed Transactions on Collaborative Computing*, 2016. (Accepted).

[59] D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Jou. on Autonomous Agents and Multiagent Systems*, 14(1), January 2007.

[60] S. R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing (ICAC04)*, May 2004.

[61] D. Wijesekera and S. Jajodia. Policy algebras for access control: the propositional case. In *Proceedings of the 8th ACM conference on Computer and communications security*, pages 38–47, 2001.

[62] Wenxuan Zhang, Constantin Serban, and Naftaly H. Minsky. Establishing global properties of multi-agent systems via local laws. In Danny Weyns, editor, *Environments for Multiagent Systems III, LNAI 4389*. Springer-Verlag, 2007.