Scientific Research Publishing

Journal of Software Engineering and Applications, 2018, 11, 28-68
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

On the Dependability of Highly Heterogeneous and Open Distributed Systems

Naftaly H. Minsky

Department of Computer Science, Rutgers University, New Brunswick, NJ, USA
Email:minsky@cs.rutgers.edu

How to cite this paper: Minsky, N.H. (2018) On the Dependability of Highly Heterogeneous and Open Distributed Systems. *Journal of Software Engineering and Applications*, **11**, 28-68.
https://doi.org/10.4236/jsea.2018.111003

Received: November 29, 2017
Accepted: January 27, 2018
Published: January 30, 2018

Copyright © 2018 by author and ScientificResearch Publishing Inc.
This work is licensed under the CreativeCommons Attribution International
License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/
Open Access

Abstract

This paper introduces an architecture of distributed systems that facilitates the implementation of a substantial range of dependable *system properties*, *i.e.*, properties that span an entire system, or a set of components dispersed throughout it. This architecture, called GDS, for *governed distributed system*, governs the system by controlling the flow of messages between its actors, independently of the internals of the interacting actors. This governance is done via an enforced collection of *interaction laws* organized into a modular and conflict free *hierarchical ensemble*. This ensemble of laws is sensitive to the history of interaction; and it is enforced in a decentralized manner, and is thus scalable. The dependable system properties that can be implemented under GDS can have the following beneficial consequences, among others: a) the ability to establish *regularities* over the system, rendering it more coherent, and easier to reason about; b) the ability to provide a degree of *trust* among the disparate actor of the system; and c) the ability to ensure compliance with interaction protocols that are essential for distributed computing. Consequently, the GDS architecture can have a significant impact on the following important system qualities: security, fault tolerance, auditability, and manageability.

Keywords

Open Distributed Systems, Governance, Dependability, System Properties, Middleware, LGI, Fault Tolerance, Security

1. Introduction

Heterogeneous distributed systems suffer from considerable difficulties in establishing *system properties* in a *dependable* manner. By a "system property", we mean a property that spans an entire system *S*, or a set of *actors*[1] dispersed

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

throughout it. Such a property may, for example, be: a) that certain types of messages sent by *any* actor of *S* are logged for audit purposes; b) that if *any* actor *x* of *S* gets a message "stop" from a system operator, *x* would stop communicating with all other actors of *S*, except for operators; and c) that all members of a group of actors dispersed throughout *S* comply with a given interaction protocol—see detail of this example below.

And we consider a system property *P* of a system *S* to be *dependable* if it satisfies the following, broadly stated, conditions: 1) it is reasonably easy to verify that *P* is satisfied by *S*; and 2) *P* is stable with respect to changes of the code of *S*—that is, it is not likely to be violated by most changes of the code of various system components, or by the unpredictable behavior of other kind of actors, like people. The dependability of system properties are essential for important qualities of a system, such as fault tolerance, security, auditability, and manageability—which are key aspects of the dependability of a system as a whole [1].

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] [2] [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. Many of the most critical societal systems and infrastructures are at least partially open. These include grids, federated organizations of various kinds, supply chains, and various systems-of-systems. Other systems tend to become progressively open, as they grow in size and evolve. Moreover, some systems are designed to be open, with the hope that this would make them more flexible. The concept of *service oriented architecture* [4] (SOA) is a prominent example of this trend, which is being adopted by a wide range of complex distributed systems. The following example illustrates the difficulty in establishing dependable system properties of heterogeneous and open systems.

Consider, for example, a system *S* that contains a distributed database, consisting of 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 a necessary condition for ensuring *serializability*—an important criterion of correctness of concurrent transactions—is that every client observes the *two-phase locking* (2PL) protocol [5], defined as follows:

The (2*PL*) *protocol*: *Once a client releases one of its locks*, *it does not acquire other locks until all its locks are released*.

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 problem is, how can such a protocol be implemented dependently.

On the Conventional Approaches for Establishing Dependable System

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

Properties in a Heterogeneous Distributed System: The seemingly natural approach for establishing system properties—such as the 2PL protocol—is *code based*. For example, to ensure that every potential client of a distributed database observes the 2PL protocol, each of them is to be programmed to comply with it. But doing so everywhere in an open system, is laborious, error prone, and hard to verify—virtually impossible to verify in an open system. 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 all clients use this stub, and that none of them modifies it to gain some advantage? Moreover, the use of such a stub is problematic when clients are programmed in different languages.

It has been suggested (see [4] for example) that one can rely on strong management and good software engineering practices to address these kinds of problems. But while such, mostly human, disciplines are clearly necessary, they cannot cope reliably with the complexities of a large open system—due to lack of global knowledge of, and control over, the code of its component parts. *We need a more rigorous and more reliable solution for the problem of dependability of highly heterogeneous distributed system*.

Our Approach: The approach of this paper to the problem in question is based on the following observation: If a system property of an open system is based on the code of many of its actors it cannot be dependable without the knowledge of, and the control over, all this code. But since such knowledge and control of many actors of an open system is not generally available, we adopt here a *level of abstraction* that ignores the internals of the actors—focusing on the observable, messaging-based, interaction between them. And we provide means for controlling the interaction between the actors of a system by specifying a *law* about how messages should flow between its actors, and between them and the outside world; and by enforcing this law via a suitable middleware.

We call such an enforced policy the *fabric* of the system, generally denoted by F, as it can be made to establish important aspects of the *underlying structure* of the system so governed[3]. A system governed by such a fabric is called a *governed distributed system*, or GDS. As we shall see, the existence of a fabric enables the dependable implementation of many useful system properties. Moreover, although the fabric itself is oblivious of the behavior of all the actors of a given system, it is able to establish dependable system properties that are contingent on the knowledge of one or few actors (cf. Section 4.3.1).

It should be pointed out that controlling the flow of messages in order to govern a distributed system is far from a new. It is what conventional access control (AC) mechanisms does, and it is behind the concept of "policy based systems", and of other types of system architectures—as discussed briefly in Section

[3]We employ here one of the dictionary (the American Heritage) definitions of the term "fabric" as "an underlying structure" of a complex system.

6. But these attempts at the governance of a distributed system, do not satisfy the design principles of GDS stated in Section 2—which are necessary for the effective implementation of dependable system properties.

The Structure of this Paper: The rest of this paper is organized as follows: Section 2 spells out our design principles of the GDS architecture; Section 3 is an outline of a middleware that facilitates the fulfillment of these principles, and which serves as a basis for GDS—this middleware, called LGI, has been published widely, and is outlined here to make this paper reasonably self-contained; Section 4 introduces the concept of GDS, and discusses its key properties, including the means provided by GDS for establishing system properties dependably. This section also provides an example of a GDS, which is an outline of an implemented case study of it; Section 5 discusses the potential impact of GDS on various aspects of distributed systems—focusing on the *fault tolerance* at the application level of distributed systems, and on their security; Section 6 discusses related work; Section 7 lists several open problems, raised by the GDS architecture, that need to be addressed for this architecture to attain its full potential; Section 8 concludes this paper.

Finally, it should be pointed out that this paper provides only a *proof of concept* for GDS, as its practical effectiveness is yet to be validated by its use for real complex application—which requires industrial participation.

2. The Design Principles of GDS

We outline here some of the key properties we require a GDS to satisfy; they pertain to the nature of the fabric of a GDS, and to the middleware used for enforcing it. Each of these requirements is followed by its rationale.

R1: The governance of a GDS should be sensitive to the history of interaction—*i.e.*, it should be *stateful*—and it should enable a degree of *proactive control*. Statefulness is required, in particular, for establishing interaction protocols, such as our example 2PL protocol. Proactive control is required for ensuring a degree of *liveness*.

R2: The enforcement policies by the fabric of a GDS should be *decentralized* and *strict*: Enforcement should be *decentralized* for the following reasons: a) for scalability, which cannot be achieved with a centralized—even if replicated—reference monitors, under highly stateful control (as shown in [6]); and b) for preventing the enforcement mechanism itself from becoming a single point of failure, and a central target for attacks.

And the enforcement of policies should be *strict* for the sake of dependability. By strict enforcement we mean, for example, that messages prohibited by the fabric should be blocked. (Note, however, that a policy established by the fabric may itself be *lax* in various respects. For example, the fabric may allow some undesirable messages to be transferred, requiring only that they would be logged for future disposition—such a lax provisions of a fabric should itself be strictly enforced.)

R3: The fabric of a GDS should be *modular*: Modularity of the fabric is required because a complex distributed system, such as one that supports a federated enterprise or a grid, cannot be governed effectively via a singleton monolithic policy. Rather, its governance is likely to require a whole ensemble of semi-independent policies, which we call *laws*. For example, consider a system that is partitioned into several semi-independent divisions, belonging to different administrative domains. Such a system, as a whole, needs to be governed by a *global law*. But each of its divisions is likely 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 such a system, possibly crosscutting through several of its divisions, may have to interact with each other subject to some coordination law. There may be many such crosscutting laws in the fabric of a system, and all of them must conform to its global law. Furthermore, the various laws that collectively govern a given system may be defined by different stakeholders, with little or no coordination which each other—and these stakeholders may themselves belong to different administrative domains.

By modularity of the fabric F that consists of such a collection of laws we mean, in particular, that F should be *free from inconsistencies* between the laws it consist of. This would facilitate the incremental construction of the fabric by different stakeholder, and would make the fabric simpler, and easier to reason about.

R4: The evolution of the fabric of a GDS should be *controllable*: This is required because a haphazard change of the fabric might have a damaging effect on the system it governs. So, we need the ability to regulate who can change the fabric, in which manner, and under which circumstances.

3. On the Middleware Used by the GDS Architecture

The GDS architecture requires a suitable middleware for formulating the policies of the fabric and for enforcing them. We have chosen for this purpose a middleware called *law governed interaction* (LGI), mostly because it satisfies requirements R1 and R2 above, and it helps in satisfying requirements R3 and R4. We outline here the structure of this middleware and some of its key properties, to make this paper reasonably self contained. This section can be skipped by a reader who is familiar with LGI.

Although LGI is broadly related to the concept of system-centric access control (AC)—exemplified by RBAC [7], Ponder [8] and XACML [9]—it is very different from it both structurally and functionally. Due to the unusual nature of LGI, its outline here may not be sufficient to convince the reader of the veracity of our claims about it. But LGI has been published widely and some of these publications are cited in due course. The manual of LGI may also be useful in this respect. In fact, what is discussed in this section is a basic version of LGI that has been designed for systems that are simple enough to be governed by a *single law*. A major enhancement of this middleware that has been developed for com-

plex systems is described in Section 4.

The rest of this section is organized as follows: Section 3.1 describe the gist of the LGI middleware; Section 3.2 describes the concept of LGI laws, which is related to, but very different from, the concept of policy under AC; Section 3.3 introduce an example of an LGI law; Section 3.4 discusses the expressive power of LGI; Section 3.5, outlines the concept of *conformance hierarchy* of laws, which facilitates the modular organization of the fabric of a GDS, as stipulated by requirement R3 in Section 2; and Section 3.6 discusses the trustworthiness of LGI, and its performance.

3.1. The Gist of LGI

As pointed out above, we are dealing here with systems that are simple enough to be governed by a single monolithic law. Such a law is denoted by $\mathcal{L}$, and the system governed by it is called an *$\mathcal{L}$-community*, or simply a *community*. Now, 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 not do so directly. Rather, LGI controls the *interactive activities* of each actor c of an $\mathcal{L}$-community C, subject to 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.

This localization of control—which, as argued in sway does not reduce the expressive power 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 it must interact with other members of C via a *private controller*, denoted by $T_c^{\mathcal{L}}$, which is trusted—for reasons to be discussed later—to mediate the interactive activities of c, subject to law $\mathcal{L}$. $T_c^{\mathcal{L}}$ runs on a *generic controller* designed to interpret and enforce any well formed law loaded into it. $T_c^{\mathcal{L}}$ can maintain a state (sometime called *control state*) that represents some function, determined by law $\mathcal{L}$, of the history of the interaction of c with the rest of the system. The formation of this state is regulated by law $\mathcal{L}$, and the ruling of the law for an interactive event that occurs at $T_c^{\mathcal{L}}$ can take its local state into account. (Note that there is no limit on the size of the state of a controller, which is not a finite state machine.)

The pair $\langle c, T_c^{\mathcal{L}} \rangle$ is called an *$\mathcal{L}$-agent*, or simply and agent, which is said to be *animated* by the actor c. (We often denote an agent such as above by the name c of its actor, expecting the potential ambiguity to be resolved by the context.) **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 the actor that animates it.) 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, but they are likely to have different states due to their different history of interaction.

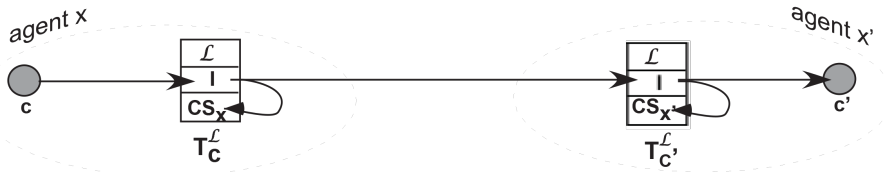

Figure 1. Interaction between a pair of LGI-agents, mediated by a pair of controllersunder possibly different laws.

A key property of LGI-based communication is that a pair of -agents can recognize each other as such. 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 4.3.6.)

3.2. 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 the three elements E, S_x, and O outlined below (see [10]).

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 S_x of agent x:* 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.

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

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

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 atomically 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.

1) 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.

2) Besides changing the state of the operating agent, the law can cause some messages sent by its actor to be changed, rerouted, or blocked; and it can initiate messages proactively. Moreover, the law can respond to various exceptions that may occur at a given agent.

3) 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 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—and another, safer, law-language is under consideration.

3.3. An Example of an LGI Law

The purpose of this section is to provide a taste of the nature of LGI laws. We do this by introducing in details a simple law that establishes a somewhat contrived coordination protocol, which illustrates some key characteristics of LGI. This protocol, which we call *ping-pong* protocol, may be viewed as representing *polite conversation*, as will become apparent below. For another example the reader is referred to [11], which introduces the $\mathcal{L}_{2PL}$ law that implements the 2PL protocol described in Section 1.

The ping-pong protocol is defined as follows: Consider a group G of actors that interact with each other subject to this protocol. Members of this group can exchange two kinds of messages: *ping* messages, used for posing questions, or requests; and *pong* messages, used as a reply to previously received ping messages. Such exchange is subject to the following constraint: once a member x of G sends a ping message to another member y of G, x would not be able to send other pings to y until it gets a reply from y in the form of a pong messages. And y can send only one pong message to x for every ping it gets from it. (The sense in which such exchange may be considered polite is fairly self evident.)

Figure 2 displays a law $\mathcal{L}_{PP}$ that establishes this protocol over the group G of actors that chose to operate under it. This law is written in the Prolog-based law-language, which can be understood, roughly, as a set of event-condition-action (ECA) rules. Since some readers may not be familiar with Prolog, and with this law-language, we provide the following explanation for this law.

First, under this law the ping and pong messages are represented by ping(M) and pong(M), respectively, where M is an arbitrary text. Also, this law uses two kinds of terms in the control-state (CS) of every agent $x$in G, to facilitate the control over its interaction with other agents: a) the term *pinged to* (y) is

```
Preamble: law(PP,language(prolog)).

R1. sent(X,ping(M),Y)
        :- not(pingedTo(Y)@CS), do(add(pingedTo(Y)),do(forward).

R2. arrived(X,ping(M),Y) :- do(add(pingedFrom(X))), do(deliver).

R3. sent(X,pong(M),Y)
        :- pingedFrom(Y)@CS, do(remove(pingedFrom(Y))),do(forward).

R4. arrived(X,pong(M),Y) :- do(remove(pingedTo(X)@CS)), do(deliver).
```

Figure 2. The ping-pong law $\mathcal{L}_{PP}$.

intended to mean that *x* pinged *y*, and did not yet get a pong as a reply to it; and (b) the term *pinged form* (*y*) is intended to mean that *x* got a ping from *y*, and did not yet reply to it by a pong. We can now explain the effect of each of the four rules of law $\mathcal{L}_{PP}$, as follows.

By Rule *R*1 of this law, any agent *x* in *G* can send a ping to any *y* in *G*, provided that *x* does not have a term *pinged to* (*y*) in its CS. And the sending of a ping to *y* would add the term v *pinged to* (*y*) to the CS *x*. When this ping arrives at its destination *y* it would, by Rule *R*2, add the term *pinged form* (*x*) to the CS of *y*, and the ping itself would be delivered to the actor of*y*.

Now, by Rule *R*3, any agent *x* can send a pong to any *y*, provided that it does have a term *pinged form* (*y*) in its CS. If this is the case, then this term would be removed from its CS, and the pong would be forwarded. When this pong arrives at its destination *y* it would, by Rule *R*4, remove the term *pinged to* (*x*) to the CS of *y*, allowing *y* to send to *x* another ping, and the pong would be delivered to the actor of *y*.

It should also be pointed out that an actor communication subject to law $\mathcal{L}_{PP}$ can communicate only with actors that also communicate under the same law. This is implicit in the structure of the law displayed in **Figure 2**, which is apparent for anybody familiar with this law-language.

3.4. On the Expressive Power of the Local LGI Laws, and on Their Global Sway

One may suspect that the decentralization of control under LGI, and its strictly local laws, involves a reduction of its expressive power. But this is not the case, in the following sense: Consider a centralized control mechanism (CCM) that mediated all the interaction between a given group of actors in*G*, thus having a global view of the interaction between all the actors of the system. Suppose also that CCM is stateful, in the sense that it can maintain an arbitrary function of the entire history of the interaction between all the members of *G*. It turns out that any policy that can be established by CCM can also be established under a local LGI law. This has been shown in [12] (Section 4), simply by simulating any given CCM-based policy, via a local LGI law.

Of course, this is a theoretical construction, which is not intended for practical use. But, as we shall see in paragraph 2 below, similar constructions can be used

for the implementation of global aggregate properties which are sometimes required in a given system.

On the Global Sway of Local LGI laws: We distinguish between two kinds of global system properties, which we call *regularities*, and *aggregate properties*. We define them below, and show how they can be established over a distributed community.

1) *By the term regularity* we mean here *compliance with a fixed principle*[4]. Formally, a regularity of a given $\mathcal{L}$-community C can be expressed as a universally quantified formula of the form $\forall x \in CP(x)$, where $P(x)$ is a local property of agent x of C—that is, $P(x)$ is defined over the local state of x, along with an event that occurred at it at a given moment in time. Such regularities are the direct consequences of LGI, due to the fact that all members of an L-community comply with the same law $\mathcal{L}$. An example of such regularity is the compliance with the Ping-Pong protocol by all members of the $\mathcal{L}_{PP}$-community.

2) *By aggregate property* we mean a property that depends on the state and behavior of all members of the community in question, or on a substantial part of its membership. As an example, consider an $\mathcal{L}$-community C whose members should be able to issue purchase orders for any amount of money, provided that the following aggregate property Q is satisfied: *The total amount of money thus paid by members of this community does not exceed the limit U*. Obviously, this property cannot be established directly by local constraints on the purchases of each member of C. But this aggregate property can be established indirectly by a strictly local LGI law, as follows.

Instead, of allowing every member of community C to issue a purchase order, our law $\mathcal{L}$ would provide the power to do so to one actor—call it BO (for *budget office*)—which is to do so on behalf of the other actors of C upon their request. This can be done in the following way. Let law $\mathcal{L}$ of C have two different modes of operations, depending upon the operating agent, as follows. First, when $\mathcal{L}$ resides in a controller associated with any member c of C except BO, $\mathcal{L}$ operates as follows: whenever c sends a purchase order to some vendor v, the law causes this message to be rerouted to BO. Second, if $\mathcal{L}$ resides in a controller associated with BO, it operates as follows: it forwards any purchase order it gets to its intended destination, *but only if* the total amount of money spent resulting from this purchase would not exceed U. (Basically, this is the way that most enterprises operate.)

Note that law $\mathcal{L}$ described above is strictly local. It is true that implementing our aggregate property require some centralization in BO, and some overhead. But this is involved just with purchase orders. And as long as such orders constitute a small fraction of the total volume of interactions among members of C, such partial 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

[4]See the American Heritage Dictionary.

we have implemented in [13]. But our experience with LGI suggests that the implementation of most useful aggregate properties have only a marginal effect on the efficiently and scalability of a community.

3.5. The Organization of Laws into a Conformance Hierarchy

LGI enables the organization of a set of laws that collectively governs a single system, into a coherent ensemble 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 (cf. Section 1). 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 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 [6]; here we provide just an informal introduction of this concept.

The Nature of Conformance of LGI-Laws: Several access control mechanisms [9] [14] 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. 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 Section 3.2. The meta part of $\mathcal{L}$ circumscribes the extent to which laws subordinate to it are allowed to deviate from its ground part. 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.

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 in question to law $\mathcal{L}_R$. $\mathcal{L}_R$, in turn, 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 $\mathcal{L}_R$.

3.6. On the Implementation of Controllers, and on Their Performance

We mention here briefly two different ways for controllers to be implemented. One way is to have a trustworthy organization, called controller service (CoS), whose business it is to rent the use of genuine controllers to its customers. Such controllers would authenticity themselves by digital certificate signed by a certification authority of the CoS. The CoS would maintain a collection of servers each of which will host a number (usually in the hundreds) of independent controllers. Such a CoS could provide its controllers to arbitrary clients, or be dedicated to a single organization, as will be the case in this paper (cf. Section 4.1).

There are other ways for implementing trusted controllers. For example actors operating via their smart phones may have their private controllers built into their phones. We have done that, experimentally, in [15], and one can make such controller quite secure using a TPM-like technology, which on Android, may be TrustZone [16]. Also, if one has a secure co-processor [17] attached to one's host, one can implement a controller on it.

Regarding performance: A comprehensive study of the overhead incurred by LGI control had been published in [12]. 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.

4. The Concept of Governed Distributed System (GDS)

We start in Section 4.1 with a definition of the concept of GDS. Then we proceed as follows: Section 4.2 illustrates this concept via an outlines of an implemented example; and Section 4.3 introduces the main key aspects of GDS.[5]

4.1. The Definition of GDS

We define a GDS, denoted here by S, as a four-tuple

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

where Ac is a set of *actors* that belong to S—in a sense to be clarified later; 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 is a controller-service (CoS) which is assumed to be maintained by the organization that manages S, and can be viewed as the trusted computing base (TCB) [18] of S. In other words, T provides the set of LGI-controllers that enforce the laws in F. We elaborate on these elements of GDS, and on its overall architecture, in the following numbered paragraphs. We then discuss the process of the construction of a GDS.

1) **The set Ac of actors:** This set consists of all the actors that communicate subject to laws of the fabric F of S. We make here several comments about Ac. First, although most of the actors in Ac are likely to be software components, some of them may be people operating via their smart-phones, or such; and some may be physical devices, like sensors and actuators. GDS treats all such actors uniformly.

Second, a given member c of Ac may not belong exclusively 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. This is because we adopt a level of abstraction that ignores 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 *rogue communication*, and the consequences of such communication to the concept of GDS are discussed in Section 4.3.8. (Note: The term "rogue" means here unprincipled, and not bad or dishonest; indeed, as we shall see later, rogue communication is sometimes beneficial.)

And third, this architecture is silent on the membership of Ac, except that Ac is assumed to contain one distinguished actor with a specified role. It is called the *fabric server* (or, F-server for short), and its role is to maintain the fabric F of this system. For more about the F-server see paragraph 3 below.

2) **The fabric F:** We make here two comments about F: First, 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 $\mathcal{L}_R$ de-

[5]Note that we are using the term GDS in two related but different senses: a) as an architectural concept, like in "GDS is particularly useful for open systems"; and as a system that satisfies this architecture, like in "a GDS Sserves a university grid...".

fined as irreversible. Other provisions of $\mathcal{L}_R$ may be modified by subordinate laws, subject to constraints imposed by $\mathcal{L}_R$ on deviations from it. In a sense, then, law $\mathcal{L}_R$ governs, directly or indirectly, the entire fabric *F*, and thus, the entire system *S*.

Second, although 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, the root law of our Acme example Section 4.2 is based on some assumptions about a specific certification authority, and about the types of certificates it signs.

3) **The Reason and Consequences of Having the Entire Fabric *F* Maintained in a Single *F*-Server:** The main reason, besides convenience, for maintaining all the laws of the fabric *F* of a given GDS *S* in the *F*-server of *S*—which is itself an actor of *S*—is for enabling effective control over the evolution of *F*, as discussed in Section 4.3.3. But the central maintenance of *F* seems to raise the following concerns: a) will this centralization reduce the scalability of GDS? b) will the *F*-server be a single point of failure of a GDS? and c) will it make a GDS more vulnerable to attacks? Fortunately, the answer to all these questions is negative, as we argue below.

First, the centralization of V has a negligible effect on the scalability of a GDS, because the *F*-server does not participate in the dynamic control over communications—this is done by the controllers, operating subject to their local laws, acquired from the *F*-server. The *F*-server is just a library of the laws in *F*, from which individual actors acquire the laws under which they are to operate—and this constitute a relatively infrequent access to the *F*-server.

Second, the fabric is likely to be modified relatively infrequently, which enables the *F*-service to be easily replicated, and thus not be a single point of failure. And third, 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 one-way hashes of the various replicas, replacing the compromised replica with a correct one.

4) **The set *Ag* of *F*-agents:** Let us first clarify some notations: 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 3.1) $\langle c, T_c^{\mathcal{L}} \rangle$—namely, an actor *c* paired with the controller it adopted, under some lawin *F*. And 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*. Also, a law can determine which actors may operate under it, in particular, by

requiring actors to authenticate themselves in some way.

5) **On the overall structure of a GDS:** It is clear that a GDS-based system S generally consists of several, perhaps many, communities each operating under one of the laws of its fabric$F$. Different such communities may overlap, in the sense that a single actor can animate agents operating under different laws—as we have seen in (4) above—and thus belonging to different communities. Also, as we shall see in Section 4.3.6, agents belonging to different communities may be able to *interoperate*, if they are allowed to do so by the fabric.

A schematic depiction of a GDS is portrayed 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; and note that these controllers will be operating subject to various laws in F. 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.

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 issue of the complete conversion of a legacy system into a GDS. However, an incremental conversion can be carried out by means of the grafting of some aspects of GDS into an otherwise conventional system—which is discussed here as well.

The Construction of a GDS from Scratch: The construction of a new system Sas a GDS can be carried out in two consecutive steps: 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 the law in F—or several such laws—to operate under, a law may impose constraints on the type of actors that may operate under it. In particular, law $\mathcal{L}_R$ of Acme—our example-system—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 by a given actor is a voluntary act—it is not enforced by GDS architecture, nor is it enforceable. Yet, an actor

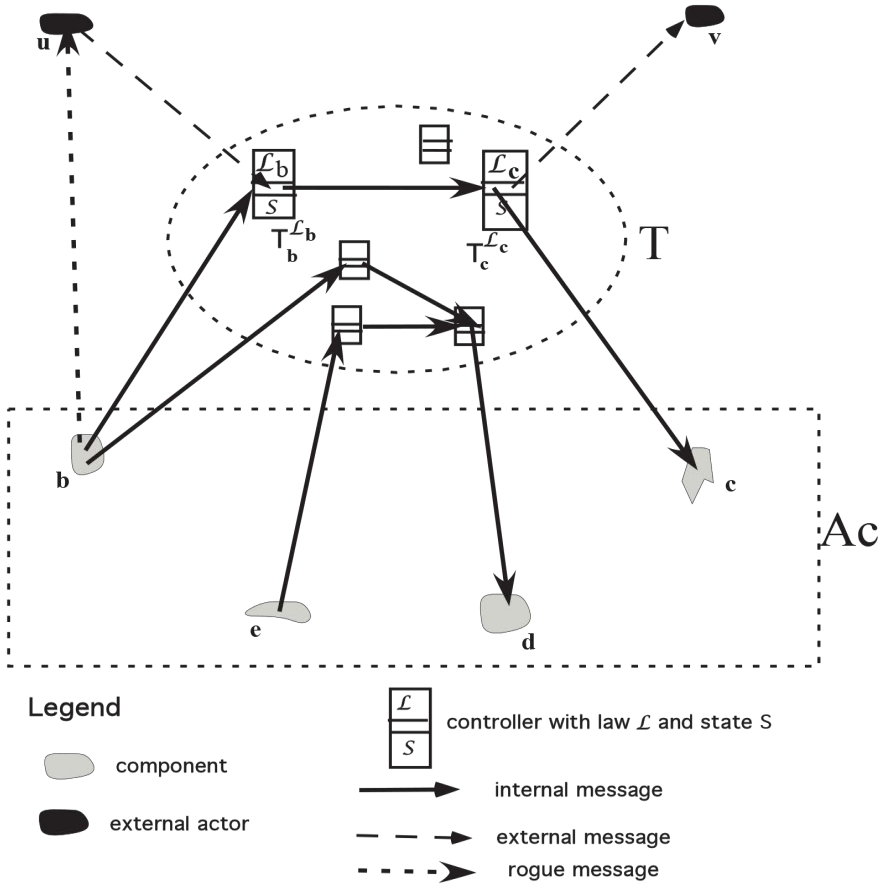

Figure 3. A schematic depiction of a GDS.

may be *virtually compelled* to operate under a certain law in F if it needs some services provided only under this law. For instance, if the databases of our 2PL example in intro 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 ensure that the members of a group G of actors comply with a given protocol defined by a law $\mathcal{L}$, when involved in a certain activity, one can have the members of G adopt this law, and operate subject to it when participating in this activity. This would create, what we have called in Section 3 $\mathcal{L}$-community. For instance, the database servers and their clients of the example in Section 1 would adopt law $\mathcal{L}_{2PL}$. There may be many such governed communities in a given system, whose laws form a kind of fragmented fabric, which may be organized into several conformance hierarchies that does not govern the entire system.

4.2. 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. $D0$ includes, in particular, the distinguished F-server, and a certification authority called AcmeCA.

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 4.2.1—consists of the root law $\mathcal{L}_R$. The second level contains the laws of the three divisions of the system, introduced in Section 4.2.2; as well as some *crosscutting laws* that govern groups of actors whose members may belong to several divisions. The placement in F of such crosscutting laws is discussed inin Section 4.3.7. The third level, introduced inin Section 4.2.3, 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 its superior law. To see how such laws are actually written see [6] 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.)

4.2.1. 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 following aspects of c: its unique name, with respect to the Acme system; its role in the system; and the division to

[6]The concepts of *delta*, and of the *meta* part of a law (mentioned below) have been introduced in Section 3.5.

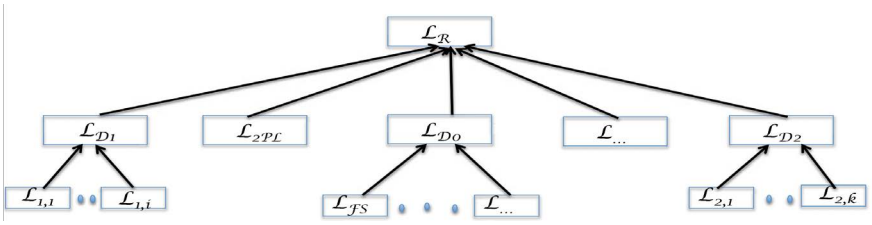

Figure 4.A basic hierarchical law-ensemble for the acme system.

which it belongs. This authenticated attributes 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 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 unique name, role, and division of its sender, with respect to the Acme system.

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 system 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, probably a person, that has a role of an operator—would lose the ability to send or receive *F*-messages of the specified pattern; and if the pattern is "all", then it would lose the power to send and receive all *F*-messages.

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

R-auth provides Acme with a degree of control 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 attributes 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 with an official identifier of its sender. The identifier in question is more informative and more trustworthy than identifying the sender by its IP address. This is because an IP address can be spoofed; and even if not spoofed, it does not carry much meaning to the receiver. Note that keeping this identifier in the state of every *F*-agent can be useful in many ways. In particular it is used here for enforcing the constraints of Rule 3.

The closest thing to this measure in the literature is a single sign on (SSO) mechanism [19]. But, while enabling identification, SSO does not ensure that it always happens. Moreover, SSO is a centralized mechanism, with all the disad-

vantages of centralization.

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 4.2.2. (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. This is just an example of the ability of the fabric to establish dependable monitoring, and facilitate the audit, of message flow.

Rule 5 enables system operators to prohibit any given *F*-agent from sending 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.

4.2.2. Division Laws

A division law, say law $\mathcal{L}_{D1}$ of division *D*1, is to be derived from the root law 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 D*1*:* Given that Rule 1 of law 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 *D*1.

2) *Imposing Constraints over Intra-Division Interaction:* Recall that all intra-division interactions have been permitted by 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 *D*1.

3) *Enabling Selected Inter-Division Interactions:* Recall that inter-division interactions are prohibited by law R, as a default, allowing subordinates laws to permit them. Note, however, that for an interaction between two divisions, say *D*1 and *D*2, 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 *x*1 in *D*1 to an *F*-agent *x*2 in *D*2, law $\mathcal{L}_{D1}$ needs to permit *x*1 to send a message to *x*2, and $\mathcal{L}_{D2}$ needs to permit *x*2 to receive this message. Of course, such a permission may be formulated so it applies to whole sets of interaction types; thus, 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.

4.2.3. 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 its division. Some such reasons are discussed below.

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—unless *c* is highly reputed. This is particularly true 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 (cf. Section 4.3.5), 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 we elaborate on 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* [20]. 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, in particular, 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 [21].)

4.3. Key Aspects of GDS

We discuss in this section several key aspects of the GDS architecture, as follows: 1) the means provided by GDS for establishing system properties dependably; 2) freedom from inconsistencies between the laws of a fabric; 3) regulating the

evolution of the fabrics of a GDS; 4) perspective on the strict enforcement of the laws; 5) law-based trust: a behavioral-trust modality induced by GDS; 6) flexible interoperability; 7) the handling of crosscutting laws; 8) rogue communication, and its limited effect on a GDS.

4.3.1. The Means Provided by GDS for Establishing System Properties Dependably

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 *perfect F*-properties and *contingent* ones, as stated below. After making this distinction we discuss the sense in which such system properties can be considered dependable.

Perfect and Contingent *F*-properties: 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*. The root law of our Acme case study (cf. Section 4.2.1) establishes several such properties. One of them, due to Rule 4, establishes the logging of all inter-division messages. Another example of a perfect *F*-property is the compliance with the 2PL protocol by all the clients of the databases that operate under law $\mathcal{L}_{2PL}$.

A *contingent F-property* is one established by *F*, *under the assumption* that one, or relatively few, actors satisfy a specified condition. One example of a contingent *F*-property—established by Rule 2 in law $\mathcal{L}_R$ of Acme—is the identification of the sender of any message, via its correct name with respect to Acme. This property is satisfied only under the assumption that the certification authority *AcmeCA* provides all members of *Ac* with their correct names in the certificates it signs for them. Another example of a contingent *F*-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 needs to ensure that the few individual databases in question satisfy the following conditions: a) that they serve only clients that operate subject to law $\mathcal{L}_{2PL}$; and b) that they deal correctly with requests to lock and unlock database items.

On the Dependability of *F*-Properties: We have defined in intro the dependability of a system property in terms of the ease of its verification, and in terms of its stability. A perfect *F*-property satisfies this definition in the following manner[7]:

1) *Ease of verification:* It is generally easier to verify a system property *P* if it is a perfect *F*-property, than if it is implemented by the code of all or many system actors. This is because *F* is likely to be orders of magnitude smaller than the code of all system actors; and because of the lack of central knowledge of the code of all actors.

[7]We admit that the following arguments about the dependability of *F*-properties are somewhat soft perhaps necessarily so, because the very concept of dependability is soft.

2) *Stability:* Since F is oblivious of the code of the system, it follows that a perfect F-property P is invariant under changes of this code. Moreover, although an F-property can be changed by changing F itself, the evolution of F can be regulated (cf. Section 4.3.3). So one would be able to depend on P not to change haphazardly during the evolutionary lifetime of the system. Moreover we expect the evolution of F to be relatively slow.

These modes of dependability, justified above for perfect F-properties, can be viewed as being *approximately satisfied* for contingent F-properties—and the less one assumes about the code of actors, the better approximation it is. Henceforth we will refer, somewhat loosely, to F-properties as dependable, whether they are perfect or contingent.

4.3.2. Freedom from Inconsistencies between the Laws of a Fabric

A system governed by a collection of laws, or policies, may suffer from inconsistencies, particularly if these laws are formulated by different stakeholders. Indeed, many access control mechanisms are plagued by the *policy inconsistency problem* [9] [22]. The conventional remedy to this problem is to apply various *conflict resolution techniques* for resolving such inconsistencies. But as has been shown in [23], 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 simplifies the reasoning about it.

Of course, consistency does not mean correctness. A fabric, and any of its laws, can be wrong in a sense that it does not do what its writer, or writers, intended. Dealing with the correctness problem—by reasoning about the fabric, or by testing it—is beyond the scope of this paper. But, the fabric is expected to be orders of magnitude smaller than the system it governs, so that informal reasoning about it may be sufficient.

4.3.3. Regulating the Evolution of the Fabrics of a GDS

Given the sway held by the fabric over the behavior of the system governed by it, 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$F$, what kind of changes can one make, and under which circumstances. 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 independently by different stakeholders then it may be necessary to establish a coordination protocol between them.

This type of regulation can be done in the following way. As stated in Section 4.1, the fabric F of a given GDS is maintained in the F-server, which is itself an actor of the system in question, and thus must operate subject to some law in F—let this law be called $\mathcal{L}_{FS}$. Now, since changes of F must be carried out by means of messages sent to the F-server, and since these messages are governed by F—directly by its own law $\mathcal{L}_{FS}$—it follows that F can regulate its own evolution.

This *self regulatory* nature of the fabric is an essential property of the GDS architecture. And it does not contradict the fundamental unpredictability of the long term evolution of software systems. Because the law that regulate the evolution of the fabric can be designed so that it can itself be changed, presumably by authorized stakeholders.

4.3.4. Perspective on the Strict Enforcement of Laws

Laws are strictly enforced by the LGI middleware, rather than 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, in particular by most multi-agent systems, such as OMNI [24]. This is because strict enforcement is essential for dependability. For example, if a message is considered harmful to its recipient it should be blocked, rather than reported for future disposition—after the harm was done. As another 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, such *report & sanction* approach to law enforcement is sometime necessary. In particular, because it is not always possible to determine locally 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 responded to via a relatively suitable sanction—even if this is done long after the offending event happened. This is generally how most social laws are being enforced.

Fortunately such *report & sanction* approach to law enforcement can be easily facilitated under GDS, with its strict enforcement of laws. This can be done simply by writing lax laws, but enforcing them strictly. For example, instead of blocking a potentially harmful message, or changing it in any way, a law can be written to let this message pass, but to log it in some logging service. The strict enforcement of such a lax provision would ensure that the logging is done, but the disposition of such a log cannot be determined by the fabric alone.

4.3.5. 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 1 (L-trust) *The members 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 the actors that animate these agents have no trust in each other.*

Moreover, each of these agents can identify the law under which its interlocutor operates, as well as the position of this law in the hierarchical structure of the fabric.

This trust rests on the following properties of GDS: 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 one-way 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.

Note that this mode of trust is fundamental to the GDS architecture as it facilitates some of its basic features—such as flexible interoperability, as discussed below.

4.3.6. Flexible Interoperability

As pointed out in Section 4.1 (cf, paragraph 6), a GDS generally consists of several communities operating under different laws, and some members of different communities may need to *interoperate*, *i.e.*, communicate with each other, despite their different laws. We first discuss here how such interoperation is carried out, comparing it to the manner in which interoperation is done under access control. We will then comment about interoperation between the *F*-agents of a given GDS, and actors outside of this system.

Interoperation between *F*-Agents Operating Under Different Laws: To put interoperation in context, we start by reviewing how it is handled under conventional access control (AC). This issue has been addressed frequently in the AC literature [25]; generally using 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, [25]): 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 [6], 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 *x*1 and *x*2, operating under laws $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Due to the locality of LGI-laws, and the resulting 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 *x*1 and *x*2 to interoperate. Rather, each of this law may permit a degree of interoperation with the other. They may, for example, allow the exchange of only certain types of messages, under specified circumstances. Of course, each of these laws may need to know the nature of the law that its interlocutor operates under. This is enabled by L-trust. For example, consider two organizations, say CIA and FBI, each operating under its own law. But each of these laws can specify the type of interoperation it permits with the 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*; and, of course, 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 $\mathcal{L}$ under which an *F*-agent *x* operates can permit it to communicate with any actor *z* over the Internet; although $\mathcal{L}$ may impose some condition on such communication. In particular, $\mathcal{L}$ may require *z* to authenticate itself in a certain manner. Note that **Figure 3** depicts interaction between an *F*-agent *b* and an actor *u* that does not belong to the GDS in question.

4.3.7. 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 the 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 a crosscutting law is $\mathcal{L}_{2PL}$, *i.e.*, the law that established the 2PL protocol.

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.

Fortunately, 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. In particular, 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**. And if a member of group *G* wished to operate subject to law $\mathcal{L}_C$, it can do so even if it operates under $\mathcal{L}_{D1}$, say—again,

if law $\mathcal{L}_C$ accepts it.

Note that our crosscutting laws are analogous to *aspects* under *aspect oriented programming* (AOP) [26]. Like aspects, our crosscutting laws deal with, what is called, under AOP, "crosscutting concerns". Although AOP has been designed for non-distributed systems, it has been used occasionally for distributed system as well [27]. But it is not very dependable there, particularly not for highly heterogeneous systems, for obvious reasons.

4.3.8. Rogue 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 freely in "direct communication" (via TCP/IP, say). In the context of a GDS, we call such communication *rogue*, because it is not bound by the fabric of this system. Rogue communication may be between actors that belong to the *Ac*, or between them and actors outside of the GDS in question; **Figure 3** depicts rogue communication by the dotted arrow from actor *b* of *S* to the external actor *u*.

Rogue 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 rogue communication limits—although in a minor extent, as we shall see—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 rogue 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 [9] over the actors of an enterprise, does not really determine who can access whom, because actors can simply bypass this reference monitor.

Yet, *F*-properties are not made any less dependable by rogue communication; and as we shall demonstrate in Section 5 there is a wide range of *F*-properties that can be established under GDS, regardless of any rogue communication that may exist in the system—the 2PL protocol is a case in point.

Elimination of Rogue Communication: Sometimes one may want to eliminate rogue 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. This can be done 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 sys-

tems [28]. A more systematic way for doing so should be possible under *Software? Defined Networking* (SDN) [29].

5. The Impact of GDS on the Dependability of Distributed Systems

The impact that the GDS architecture is expected to have on distributed systems, and particularly on highly heterogeneous and open systems, is due to its ability to establish dependable system properties. As explained in Section 4.3.1, such properties can be established via the fabric of a given system—they are called *F*-properties, and they can be either *perfect*, depending only on the structure of the fabric itself; or *contingent* on the knowledge of the behaviour of a few actors.

There is a wide range of *F*-properties, which can be established under GDS, contributing to important qualities of a system, including: coherence, auditability, manageability, fault tolerance and security. We will start by illustrating briefly how the first three of these qualities can be established, using examples from the root law of the fabric of our Acme system described in Section 4.2.1. This is followed by a more detailed discussion of the last two qualities, namely fault tolerance and security.

Coherence: In a sense, any regularity of a system, *i.e.*, any property satisfied everywhere in it, contributes to the coherence of the system and helps in reasoning about it. One example of such a regularity is provided by Rule 1, which ensures that every actor *x* operating under fabric *F* authenticated itself in a specified manner. Also, L-trust—the trust modality provided under GDS—contributes to coherence as it provides a degree of trust between the various agents of a GDS.

Auditability: An example of auditability is provided by Rule 4, which establishes a dependable audit trail for inter-division interaction. This suggests the potential for establishing a broad range of audit policies—but only regarding the flow of messages in a system.

Manageability: An example of manageability is provided by Rule 5 which empowers designated *operators* to control the interactive behavior of *every* system agent. This, and the ability to monitor selected messages by rules such as Rule 4, suggest how distributed systems can be managed (*i.e.*, monitored and controlled) at their application level. This, in an analogy to the manner in which local networks are managed under SNMP, which cannot be applied to the application level of systems.

5.1. Software Fault Tolerance (SFT)

Back in 1975 Brian Randell argued [30] that the traditional fault tolerance techniques, designed, for what he called "hardware failures", are not sufficient for handling the many ways in which an application may fail—which are mostly due to *software failures*, at the application level of a system. This paper gave rise to what has come to be known as "software fault-tolerance", or SFT. Considerable research effort has been devoted to SFT, mostly, but not exclusively, for local

(non-distributed) systems. See [31] for a survey. According to this research, SFT-measures are to be established by incorporating failure-handling code (henceforth "SFT-code") into the code of the original software; which may have to be done, systematically, in many parts of the system. Also, means have been developed for doing so in distributed systems, including Argus [32] and Arjuna [33].

Unfortunately, such code-based SFT technique are not available, or at least not dependable, for open distributed systems, due to the lack of overall knowledge of, and control over, the code of the various system actors, some of which may be humans. *This leaves open distributed systems quite vulnerable to their software failures*.

Fortunately, as we demonstrate in this section, many useful SFT-measures can be established, dependently, via *F*-properties. Moreover, besides being necessary for open systems, such fabric-based SFT-measures can be preferable to conventional code-based SFT-techniques for distributed systems in general. This, for two main reasons: First, fabric-based SFT-measures would be more dependable than measures based entirely on the code, as they are invariant under changes of the code—or of most of it, in the case of contingent *F*-properties. Second, enacting such measures would not complicate the code because the fabric is completely separate from it.

Of course, not all conventional SFT-measures—which can be established by inserting suitable code into the various actors—can be implemented via the fabric of a GDS. For example, a fabric cannot ensure orderly checkpointing by selected components—an important basis for many conventional FT-measures. However the use of GDS may encourage the development of more modular distributed systems, which may simplify the insertion of suitable SFT code.

We describe below a sample from the range of SFT-measures that can be established via *F*-properties. Some of the measure discussed here are very simple, others are more complex; some of them deal with fairly specific situations, others are more broad spectrum. Also, all the SFT-type measures discussed here have been implemented, and some were published, as cited below. This sample is organized into the following complementary types: 1) prevention of failures; 2) detection of failures, and containment of failing actors; 3) recovery from failures. And note that we assume here that the LGI-controller used for enforcing the fabric of the system do not fail. Our approach for tolerating the failures of controllers is beyond the scope of this paper, and will be published separately.

5.1.1. Preventing Failures

Failures can sometimes be prevented by imposing a structure on a given system that helps in avoiding situations that may lead to certain types of failures. Strictly enforced *access control*—which blocks messages that can damage their receiver—is one well known type of such failure prevention in distributed systems. Here we discuss two example of such prevention, both of which are beyond the scope of conventional access control. One involves disciplines that prevent coor-

dination failures. And the other involves the regulation of system operators.

1) Preventing Coordination failures: Consider a group G of distributed actors who need to coordinate their activities, subject to a given protocol$P$. Such a protocol may be required for an effective collaboration of the members of G towards a common goal, or for their safe competition over some resources. There is a host of such protocols devised for various purposes such as *leader election*, *mutual exclusion*, the 2PL protocol (cf. Section 1), and many others. Such a coordination may fail due to the failure of any member of G to follow the required protocol P. (In an analogy, consider what may happen if a car, approaching an intersection between roads does not stop on a red light.)

The distributed systems literature, which designed many such protocols (see [34], in particular) often assumes that all participant in a given coordination activity abide voluntarily by the protocol designed for it. But such an assumption is mostly unwarranted in open systems. Under GDS, however, if a protocol P can be formulated in terms of message passing, it can often be established firmly as an F-property of the system. This is done by expressing protocol P via a law $\mathcal{L}_P$ of the fabric, which is to be employed by all members of group G for their coordination activity. This is possible because LGI-laws are sensitive to the history of interaction, and because they are proactive—that is, they can force some actions to be carried out, via a mechanism of enforced *obligation*, thus ensuring a degree of *liveness*. This is the way protocol 2PL can be established by law $\mathcal{L}_{2PL}$, as shown in [11].

2) **Preventing Failures Caused by Operator's Errors:** A careful study [35] of failures of large systems reported that a large percentage of such failures is cause by operator's errors. One way for reducing the probability of occurrence of such errors is to regulate the intervention of operators in the workings of a system, which can be done by means of its fabric. For example, a fabric can impose constraints on what any given operator can do, and on the order that certain interventions can be carried out. Also, the fabric can impose some coordination protocol between several operators that manage a single large system. As an example, the fabric can ensure that certain intervention in the system requires the approval of two or three operators. This can be done under GDS because the operators can be defined as actors of the system they are to manage. And both their power to intervene in the working of the system, and the constraints on that power, can be defined by the fabric of that system.

5.1.2. Detection of Failures, and Containment of Failing Actors

We start with some fairly broad spectrum techniques that are immediate under GDS. And then describe a simple technique for facilitating the detection of *halting failures*.

1) **An Immediate and Broad Spectrum Technique for Detection and Containment of Failures:** Faulty messages—those that violate normative communication as defined by the fabric F—can be blocked by F. This would protect system actors from receiving faulty, and perhaps damaging, messages. But the

sender *x* of a faulty message may, or may not, be dangerous in any other sense. To help in identifying dangerous actors that need to be removed from the system, the fabric may be written to log all suspicious messages. Such a log can be analyzed, and if *x* is determined by this analysis to be dangerous in some sense, it can be prevented from sending or receiving *F*-messages, and thus be effectively isolated from the system. Such isolation can be done by some manager, who obtained its power to do so from the fabric, in a manner exemplified by Rule 5 in Section 4.2.1.

2) **Facilitating the Detection of Halting 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 *x*, is to have *x* send heartbeat messages in a specified frequency to a designated monitor *m*. This would enable the monitor to conclude that *x* failed—*i.e.* died or has been disconnected by a break in the network—if *m* did not receive heartbeat messages from it for a while. The problem is how can the monitor *m* distinguish between a dead actor, and one who just fails to send heartbeat messages to it due to some bug.

This problem can be addressed, in many circumstances by establishing a dependable heartbeat discipline, for any given set *G* of *F*-agents. This can be done by having members of *G* 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 does under LGI.

5.1.3. Recovery from Failures

For a recovery mechanism to be truly useful it should exhibit degree of generality. That is, it should be able to handle a wide range *R*of failures, by a possibly heterogeneous set *Q* of actors. As argued by the author in [36], such a wide spectrum recovery mechanism require the imposition of *suitable regularities* over a given system, which would enable: a) the sensing of the behavior of all members of *Q*; and b) exerting a degree of control over the failing actors in *Q*, such as is necessary for the recovery of from all failures of type *R*. It is, of course, hard to establish such regularities over the code of highly heterogeneous distributed system. But some regularities over the flow of messages, which can be helpful for recovery, can be established via the fabric of a GDS. For example, as already pointed out, the power of an operator of the Acme system—provided by Rule 5 in Section 4.2.1—to block the ability of an arbitrary *F*-agent to communicate, is useful for many types of recoveries. We discuss here three additional examples of fabric-based regularities that can facilitate recovery from failures.

1) **Recovery from Coordination Failures:** Not all coordination failure can be prevented by the technique discussed in Section 5.1.1. Therefore, we have implemented [37] the concept *coordinated atomic actions* (CAA) [38], for heterogeneous and open distributed systems. CAA has been introduced by Randell for *local* (not distributed) systems, as a means for recovery of a broad class of coor-

dination failures. This influential [39] concept can be described, broadly, as follows: To engage in a given coordination activity, its participants "enter" a suitable CAA, which is a kind of virtual box that can control the activities of actors in it. The host CAA ensures the ACID property[8] for this activity, and provides either forward or backward recovery, if the activity fails. The original CAA mechanism has been extended to monolithic distributed systems [40]—where one can ensure that all participants entering a given CAA adhere to its constraints, by programming them accordingly. But ours is the first implementation of this concept for highly heterogeneous and even open distributed systems.

2) **Ensuring Fail-Stop Type of failures by *F*-agents:** Fail-stop type of failure is a failure where a processor halts, and never resumes its operations [41]. Such failures are easiest to recover from. For example, a failed processor can be safely replaced with an equivalent one, when one can be sure that the failed processor will not resume its operations. And, as pointed out in [42], the well known *state machine* approach to fault tolerance is more efficient and easier to carry out when a failure is fail-stop. To facilitate fault tolerance, Schlichting and Schneider [43] 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 limitations, from our viewpoint. First, it depends on the code of the processor in question, and is, therefore, not applicable to open systems. And second, this technique applies to processors, and not to application-level processes, many of which can run on a single processor.

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 4.2.1. This rule empowers operators to block the ability 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 an operator to remove it, in this sense, from the system.

It should be pointed out that a similar assurance 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.

3) **Collaborative Reconfiguration:** Recovery from failures often involves reconfiguration of a system—which is also being used for system management in general. Most current approaches to reconfiguration (see [44], for example) employ central manager, which is assumed to have sufficient knowledge of the system, as well as sufficient power over it to carry out its task. But these conditions are hard to satisfy, particularly in open systems. Indeed, for this reason and others, there is a growing realization that reconfiguration often require collaboration between distributed actors [45] [46] [47], rather than being managed cen-

[8]ACID stands for: Atomicity, Consistency, Isolation, Durability.

trally. And it is clear that collaborative reconfigurations requires its participants to operate subject to a certain common protocol. Such a protocol can often be established as an *F*-property.

For example, we have developed [48] a collaborative reconfiguration mechanism for a *token-ring protocol*—which can, in particular, remove an agent from the ring, or add a new agent to it, without having to stop the operation of the ring, and without losing or duplicating the token.

5.2. Security

Dependable system properties are indispensable for the security of distributed systems. Indeed, if there is a useful security measure to be employed, it is generally crucial for it to be employed all over the system—that is, it needs to be a system property. And it is not useful security unless it is dependable. Some simple examples of such security enhancing properties is provided by the root law of the Acme system. One such property is established by R-auth, which states that every actor, operating under any law in *F*, authenticates itself in a specified manner. Another example, established by Rule 2, provides the receiver of every message with a trustworthy and meaningful identifier of its sender. The relevance of these two simple *F*-properties to security is self evident. Below we will describe briefly two, more intricate, examples of how security can be enhanced under GDS.

5.2.1. Intrusion Detection and Prevention

The GDS architecture provides means for complementing the conventional anomaly based intrusion detection [49] (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, or both. Moreover, the fabric can enable the removal of the sender of an illegal message form the system (cf. Rule 5 in Section 4.2.1).

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.* [50] 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.

5.2.2. The Case of Distributed Hash Table (DHT)

A DHT is a decentralized lookup service used, in particular, in a variety of P2P applications such as file sharing. Unfortunately, DHT is very vulnerable to failures because its individual components are operated by the participants of the mostly heterogeneous and open P2P application in question—and they cannot all be trusted to comply with the underlying coordination protocol of the DHT. Some of the resulting vulnerabilities of DHT where discussed in [51], along with proposal for changes to the current DHT protocols to make them more secure. But changing the protocols is often not sufficient as long as one cannot trust the compliance with them.

However, the DHT protocol can be established more securely by the fabric used to govern that P2P application in question. We have done so in [52] for one of the protocols used by various versions of DHT—other such protocols can be established by the fabric in a similar manner.

6. Related Work

In a sense, this work is related to *reflection techniques* in non-distributed systems, such as *aspect oriented programming* (AOP) [53] and others. But here we review only works papers that attempt to govern distributed systems, in some sense. We will discuss several types of such works, more or less in the increasing order of their relevance to this paper.

1) **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 [54], require all system components to be written in a given language, and to employ a common programming discipline. Other efforts, like [26], apply AOP techniques to distributed systems by assuming that all system components use AOP in a coordinated manner.

2) **Governance Based on Access Control (AC):** There are many real applications and research papers, such as [55], that control the flow of messages in a system, via some kind of access control (AC) mechanism, such as XACML [9]; or RBAC [56] and its various generalizations, such as ABAC [57], and UCON [58]. 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"; and some, like xESB [59], are design specifically for open SOA-based systems.

But none of these efforts satisfy the key requirements of GDS, as spelled out in Section 2, because of some inherent limitations of the conventional AC techniques they are based on—limitation that were discussed in [60]. Specifically, conventional AC is not fully stateful, it is not decentralized, and it does not provides the means for creating a modular and conflict free analog of our fabric—as discussed in Section 4.3.2. And none of the AC mechanisms, or of the system based on them, even addresses the issue of evolution of their equivalent of our

fabrics, and they certainly do not try to control such evolution. A rare exception, regarding decentralization, is the use of distributed firewalls for regulating the flow of messages between an enterprise and the rest of the Internet [22]. But the expressive power of firewall policies is relatively weak, and there is no effective analog of our conformance hierarchy for governing the set of firewalls that protects an enterprise.

It should be pointed out, however, that there are some—not entirely successful—attempts made for addressing some of these limitations. First, two AC mechanisms [61] [62] are sensitive to the history of interaction, but they are centralized, thus not really scalable, as shown in [60]. Second, [14] [55] organize their policies into something akin to our conformance hierarchy of laws—which is the basis of the modularity of our fabric. But they differ from ours conformance hierarchy in several ways, in particular: a) the conformance provided by these papers is not inherent to the structure of the hierarchy, but needs to be verified, mostly manually; and b) like other AC mechanisms, they are not conflict free.

3) **Norms-based multi agent systems:** Much of the literature on *multi-agents systems* (MAS) [2], and particularly on the type of MAS designed to support *electronic institutions* [63], recognizes the need for such systems to be governed by laws—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 [24] [63] [64], 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, reporting them and executing suitable sanctions for them. As we have explained in Section 4.3.4 strict enforcement of laws is essential for dependability, which is why it is employed under GDS, and it enables *report & sanction* treatment of violations with suitable formulation of the laws.

We are familiar with only one MAS project—AMELI [65]—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—perhaps, without knowledge of the much earlier publication of 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*.

4) **Some assorted related works:** Several related paper that do not fit into the classification above are worth mentioning. First, our concept of fabric may be viewed as a concrete realization of the concept of *environments* of multi-agent systems, introduced in [66]. But while conceptually close to our approach, this paper does not present a specific realization of their proposal. Second, an extensive project on "law-governed systems", led by Carlos Lucena [67] has many similarities to our work on LGI—which inspired it—and to some of its applications, like this paper. However, this project does not satisfy some of the re-

quirements of GDS because it uses non-local laws, which are enforced mostly in centralized manner, as compared to our local laws, with their decentralized enforcement. We should also mention two papers that contribute to dependability indirectly, doing so in very different manner than this paper. One of them [68] provides "accountability" to open distributed systems, namely the ability to detect faulty nodes, after they have failed. The second [69], provides a mechanism for debugging distributed system.

7. Future Research

The GDS architecture 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) **Converting a Legacy System into a GDS:** Although the construction of a GDS from scratch, as described in Section 4.1.1 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. The grafting some aspects of GDS into an otherwise conventional system, discussed in Section 4.1.1, can be used for carrying out such conversion incrementally. But this is probably not the best way for converting a system as a whole, in one swoop.

2) **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 4.3.3. But there are additional issues involved with such evolution, which have not been resolved yet. These include the following, in particular: 1) How to carry out a change in a law $\mathcal{L}$ of F, given the possible dependency of $\mathcal{L}$ on another law $\mathcal{L}'$—dependency that may be due to 'being a subordinate to $\mathcal{L}$, or due to an existing interoperation between actors subject to with actors operating under $\mathcal{L}'$. And 2) 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 [70], but doing so for a multi-law fabric presents a new challenge.

8. Conclusions

This paper introduces an architecture of distributed systems that facilitates the implementation of a range of dependable *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*, governs the system by controlling the flow of messages between its actors, independently of the internal of the interacting actors. This governance is done via an enforced collection of *interaction laws* organized into a modular and conflict free *conformance hierarchy*, called the *fabric* of the system. The fabric is sensitive to the history of interaction; and it is enforced in a decentralized manner, and thus scalable and more secure than a centralized enforcement.

The dependable system properties that can be implemented under GDS can have the following beneficial consequences: a) the ability to establish *regularities* over the system, rendering it more coherent, and easier to reason about; b) the

ability to provide a degree of *trust* among the disparate actors of the system; and c) the ability to ensure compliance with coordination protocols that are essential for distributed computing. Consequently, GDS can have a significant impact on the following important system qualities: coherence, auditability, manageability, fault tolerance and security.

Finally, it is worth pointing out that this is a *work in progress*, in two respects. First, although the implemented case study of GDS, outlined in Section 4.2, constitutes a *proof of concept* of this architecture, the real usefulness and effectiveness of this architecture in practice needs to be validated by applying it to a real large scale distributed systems. Second, the GDS architecture, as introduced in this paper, raises some interesting open issues that need to be addressed, for it to attain its full potential.

Acknowledgements

I greatly benefited from many interesting discussions about this topic with Yaron Minsky and Yair Minsky.

References

[1] Avižienis, A., Laprie, J.-C. and Randell, B. (2004) Dependability and Its Threats: A Taxonomy. In: Jacquart, R., Eds., *Building the Information Society*, IFIP International Federation for Information Processing, Vol. 156, Springer, Boston, MA, 91-120. https://doi.org/10.1007/978-1-4020-8157-6_13

[2] Artikis, A., Sergot, M. and Pitt, J. (2006) Specifying Norm-Governed Computational Societies. Technical Report, Imperial College of Science Technology and Medicine, London.

[3] Bidan, C. and Issarny, V. (1998) Dealing with Multi-Policy Security in Large Open Distributed Systems. In: Quisquater, J.J., Deswarte, Y., Meadows, C. and Gollmann, D., Eds., *Computer Security—ESORICS* 98, *ESORICS* 1998, *Lecture Notes in Computer Science*, Vol. 1485, Springer, Berlin, Heidelberg, 51-66. https://doi.org/10.1007/BFb0055855

[4] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F. and Kramer, B.J. (2006) Service-Oriented Computing: A Research Roadmap. In: Cubera, F., Ed., *Service Oriented Computing* (*SOC*), Number 05462 in Dagstuhl Seminar Proceedings, Internationales Begegnungs.

[5] Tanenbaum,A., Van Renesse, R., Staveren, H., Sharp, G.J., Mullender, S.J. and Rossum, G. (1990) Experiences with the Amoeba Distributed System. *Communications of the ACM*, **33**, 46-63. https://doi.org/10.1145/96267.96281

[6] Ao, X.H. and Minsky, N.H. (2003) Flexible Regulation of Distributed Coalitions. In: Snekkenes, E. and Gollmann, D., Eds., *Computer Security—ESORICS* 2003, *Lecture Notes in Computer Science*, Vol. 2808, Springer, Berlin, Heidelberg, 39-60. https://doi.org/10.1007/978-3-540-39650-5_3

[7] Sandhu, R.S., Ferraiolo, D. and Kuhn, R. (2000) The NIST Model for Role-Based Access Control: Towards a Unified Standard. *Proceedings of ACM Workshop on Role-Based Access Control*, ACM.

[8] Damianou, N., Dulay, N., Lupu, E. and Sloman, M. (2001) The Ponder Policy Specification Language. *Proc. of Policy Worshop*, Bristol.

[9] Haeberlen, A., Kouznetsov, P. and Druschel, P. (2007) Peer Review: Practical Accountability for Distributed Systems. *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, Stevenson, Washington, 14-17 October 2007, 175-188. https://doi.org/10.1145/1323293.1294279

[10] Minsky, N.H. (2006) Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual). Rutgers. http://www.moses.rutgers.edu/

[11] Ao, X., Minsky, N., Nguyen, T. and Ungureanu, V. (2000) Law-Governed Communities Over the Internet. *Proc. of Fourth International Conference on Coordination Models and Languages*, Limassol, Cyprus, LNCS 1906, 133-147.

[12] Minsky, N.H., Ungureanu, V., Wang, W. and Zhang, J. (1996) Building Reconfiguration Primitives into the Law of a System. *Proc. of the Third International Conference on Configurable Distributed Systems* (*ICCDS*'96).

[13] Zhang, W.X., Serban, C. and Minsky, N.H. (2007) Establishing Global Properties of Multi-Agent Systems via Local Laws. In: Weyns, D., Parunak, H.V.D. and Michel, F., Eds., *Environments for Multi-Agent Systems III, E4MAS* 2006, *Lecture Notes in Computer Science*, Vol. 4389, Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-71103-2_10

[14] Belokosztolszki, A. and Moody, K. (2002) Meta-Policies for Distributed Role-Based Access Control Systems. *Proceedings of Third International Workshop on Policies for Distributed Systems and Networks*, Monterey, CA, 5-7 June 2002, 106-115. https://doi.org/10.1109/POLICY.2002.1011298

[15] Esteva, M., Rosell, B., Rodriguez-Aguilar, J.A., Josep, A. and Ameli, L. (2004) An Agent-Based Middleware for Electronic Institutions. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, Volume 1, IEEE Computer Society, 236-243.

[16] Lazouski, A., Martinelli, F. and Mori, P. (2008) A Survey of Usage Control in Computer Security. Istituto di Informática e Telemática, CNR.

[17] Smith, S.W. and Austel, V. (1998) Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors. 3*rd USENIX Workshop on Electronic Commerce*.

[18] Rushby, J.M. (1981) Design and Verification of Secure Systems. *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 14-16 December 1981, 12-21. https://doi.org/10.1145/800216.806586

[19] Inverardi, P. and Mostarda, L. (2005) A Distributed Intrusion Detection Approach for Secure Software Architecture. In: Morrison, R. and Oquendo, F., Eds., *Software Architecture, EWSA* 2005*, Lecture Notes in Computer Science*, Vol. 3527, Springer, Berlin, Heidelberg, 168-184. https://doi.org/10.1007/11494713_12

[20] Casati, F., Shan, E., Dayal, U. and Shan, M. (2003) Business-Oriented Management of Web Services. *Communications of the ACM*, **46**, 55-60. https://doi.org/10.1145/944217.944238

[21] Serban, C., Chen, Y., Zhang, W. and Minsky, N. (2008) The Concept of Decentralized and Secure Electronic Marketplace. *The Journal of Electronic Commerce Research*, **8**, 79-101. ttps://doi.org/10.1007/s10660-008-9014-0

[22] Jajodia, S., Samarati, P., Sapino, M.L. and Subrahmanian, V.S. (2001) Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, **26**, 214-260. https://doi.org/10.1145/383891.383894

[23] Chadha, R. (2006) A Cautionary Note about Policy Conflict Resolution. *Military*

Communications Conference, MILCOM 2006, IEEE, Washington, DC, 23-25 October 2006, 1-8. https://doi.org/10.1109/MILCOM.2006.302500

[24] Vázquez-Salceda, J., Dignum, V. and Dignum, F. (2005) Organizing Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, **11**, 307-360. https://doi.org/10.1007/s10458-005-1673-9

[25] Mendonca, M., Obraczka, K. and Turletti, T. (2012) The Case for Software-Defined Networking in Heterogeneous Networked Environments. *Proceedings of the* 2012 *ACM conference on CoNEXT Student Workshop*, Nice, 10-10 December 2012, 59-60. https://doi.org/10.1145/2413247.2413283

[26] Krishnan, M. (2015) Survey on Security Risks in Android OS and an Introduction to Samsung KNOX. *International Journal of Computer Science and Information Technologies*, **6**.

[27] Subotic, S., Bishop, J. and Gruner, S. (2006) Aspect-Oriented Programming for a Distributed Framework: Reviewed Article. *South African Computer Journal*, **5**, 81-89.

[28] Phan, T., He, Z.J. and Nguyen, T.D. (2006) Policies over Standard Client-Server Interactions. *Journal of Computers*, **1**.

[29] Minsky, N.H. (2003) On Conditions for Self-Healing in Distributed Software Systems. *Proceedings of the International Autonomic Computing Workshop Seattle Washington*.

[30] Randell, B. (1975) System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 220-232.

[31] Gheorghe, G., Neuhaus, S. and Crispo, B. (2010) xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: Nishigaki, M., Jøsang, A., Murayama, Y. and Marsh, S., Eds., *Trust Management IV. IFIPTM* 2010*, IFIP Advances in Information and Communication Technology*, Vol. 321, Springer, Berlin, Heidelberg, 63-78. https://doi.org/10.1007/978-3-642-13446-3_5

[32] Liu, X.Z., Guo, Z.Y., Wang, X., Chen, F.B., Lian, X.C., Tang, J., Wu, M., Kaashoek, M.F. and Zhang, Z. (2008) D3S: De-Bugging Deployed Distributed Systems. NSDI, 423-437.

[33] Shrivastava, S.K. (1995) Lessons Learned from Building and Using the Arjuna Distributed Programming System. In: Birman, K.P., Mattern, F. and Schiper, A., Eds., *Theory and Practice in Distributed Systems, Lecture Notes in Computer Science*, Vol. 938, Springer, Berlin, Heidelberg, 17-32. https://doi.org/10.1007/3-540-60042-6_2

[34] McDaniel, P. and Prakash, A. (2002) Methods and Limitations of Security Policy Reconciliation. *Proc. of the IEEE Symp on Security and Privacy*.

[35] Bianchini, R., Martin, R.P., Nagaraja, K., Nguyen, T.D. and Oliveira, F. (2005) Human-Aware Computer System Design. *Proceedings of the* 10*th Workshop on Hot Topics in Operating Systems* (*HotOS*).

[36] Minsky, N.H. (2012) Decentralized Governance of Distributed Systems via Interaction Control. In: Artikis, A., Craven, R., Kesim Çiçekli, N., Sadighi, B. and Stathis, K., Eds., *Logic Programs, Norms and Action, Lecture Notes in Computer Science*, Vol. 7360, Springer, Berlin, Heidelberg, 374-400. https://doi.org/10.1007/978-3-642-29414-3_20

[37] Wang, Z. and Minsky, N. (2014) Fault Tolerance in Heterogeneous Distributed Systems. *Proc. of the* 9*th IEEE International Workshop on Trusted Collaboration*. https://doi.org/10.4108/icst.collaboratecom.2014.257585

[38] Xu, J., Randell, B., Romanovsky, A., Rubira, C.M.F., Stroud, R.J. and Wu, Z.X. (1995) Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, FTCS-25, Digest of Papers, IEEE, 499-508.

[39] Pereira, D.P. and de Melo, A.C.V. (2010) Formalization of an Architectural Model for Exception Handling Coordination Based on CA Action Concepts. *Science of Computer Programming*, **75**, 333-349. https://doi.org/10.1016/j.scico.2009.12.006

[40] Xu, J., Romanovsky, A. and Randell, B. (1998) Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation. *Proceedings of* 18*th International Conference on Distributed Computing Systems*, IEEE, Amsterdam, 29-29 May 1998, 12-21. https://doi.org/10.1109/ICDCS.1998.679465

[41] Schneider, F.B. (1984) Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems* (*TOCS*), **2**, 145-154. https://doi.org/10.1145/190.357399

[42] Schneider, F.B. (1990) Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys* (*CSUR*), **22**, 299-319. https://doi.org/10.1145/98163.98167

[43] Schlichting, R.D and Schneider, F.B. (1983) Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems* (*TOCS*), **1**, 222-238. https://doi.org/10.1145/357369.357371

[44] Zhang, J., Cheng, B.H.C., Yang, Z.X. and McKinley, P.K. (2005) Enabling Safe Dynamic Component-Based Software Adaptation. In: de Lemos, R., Gacek, C. and Romanovsky, A., Eds., *Architecting Dependable Systems III, Lecture Notes in Computer Science*, Vol. 3549, Springer, Berlin, Heidelberg, 194-211. https://doi.org/10.1007/11556169_9

[45] Papadopoulos, G.A. and Arbab, F. (2001) Configuration and Dynamic Reconfiguration of Components Using the Coordination Paradigm. *Future Generation Computer Systems*, **17**, 1023-1038. https://doi.org/10.1016/S0167-739X(01)00043-7

[46] Zarras, A., Fredj, M., Georgantas, N. and Issarny, V. (2006) Engineering Reconfigurable Distributed Software Systems: Issues Arising for Pervasive Computing. In: Butler, M., *et al.*, Eds., *Fault-Tolerant Systems in LNCS*, Springer-Verlag, 364-386.

[47] Weyns, D., Malek, S. and Andersson, J. (2010) On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future. *Proceedings of the* 2010 *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Cape Town, South Africa, 3-4 May 2010, 84-93. https://doi.org/10.1145/1808984.1808994

[48] de Oliveira, M., Golçalves, E. and Purvis, M. (2014) Institutional Environments: A Framework for the Development of Open Multiagent Systems. In: Bazzan, A. and Pichara, K., Eds., *Advances in Artificial Intelligence-IBERAMIA* 2014, *Lecture Notes in Computer Science*, Vol. 8864, Springer, Cham, 560-571. https://doi.org/10.1007/978-3-319-12027-0_45

[49] Stillerman, M., Marceau, C. and Stillman, M. (1999) Intrusion Detection for Distributed Applications. *Communications of the ACM*, **42**, 62-69. https://doi.org/10.1145/306549.306577

[50] Sotiris I., Keromytis, A.D., Bellovin, S.M. and Smith, J.M. (2000) Implementing a Distributed Firewall. *ACM Conference on Computer and Communications Security*, 190-199.

[51] Urdaneta, G., Pierre, G. and Van Steen, M. (2011) A Survey of DHT Security Tech-

niques. *ACM Computing Surveys* (*CSUR*), **43**.

[52] Wang, Z. and Minsky, N.H. (2015) Towards Secure Distributed Hash Table. 11*th EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing*.

[53] Kiczales, G. and Mezini, M. (2005) Aspect-Oriented Programming and Modular Reasoning. *Proc. Int. Conf. Software Engineering* (*ICSE*), 49-58.

[54] Rowanhill, J.C., Varner, P.E. and Knight, J.C. (2004) Efficient Hierarchic Management for Reconfiguration of Networked Information Systems. 2004 *International Conference on Dependable Systems and Networks*, IEEE, Florence, 28 June-1 July 2004, 517-526. https://doi.org/10.1109/DSN.2004.1311921

[55] Liskov, B. (1988) Distributed Programming in Argus. *Communications of the ACM*, **31**, 300-312. https://doi.org/10.1145/42392.42399

[56] Osborn, S., Sandhu, R. and Munawer, Q. (2000) Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, **3**, 85-106. https://doi.org/10.1145/354876.354878

[57] Yuan, E. and Tong, J. Attributed Based Access Control (ABAC) for Web Services. Proceedings of 2005 IEEE International Conference on Web Services, ICWS 2005, IEEE.

[58] Lee, D., Ahn, S. and Kim, M. (2011) A Study on Hierarchical Policy Model for Managing Heterogeneous Security Systems. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D. and Apduhan, B.O., Eds., *Computational Science and Its Applications-ICCSA* 2011, *Lecture Notes in Computer Science*, Vol. 6785, Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21898-9_19

[59] Godic, S. and Moses, T. (2005) OASIS Extensible Access Control. Markup Language (XACML), Version 2. Technical report, Oasis.

[60] Minsky, N.H. and Ungureanu, V. (2000) Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. *ACM Transactions on Software Engineering and Methodology*, **9**, 273-305. https://doi.org/10.1145/352591.352592

[61] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. (2001) Getting Started with ASPECTJ. *Communications of the ACM*, **44**, 59-65. https://doi.org/10.1145/383845.383858

[62] Ribeiro, C. and Ferreira, P. (2007) A Policy-Oriented Language for Expressing Security Specifications. *International Journal of Network Security*, **5**.

[63] Dudheria, R., Trappe, W. and Minsky, N. (2010) Coordination and Control in Mobile Ubiquitous Computing Applications Using Law Governed Interaction. *Proc. of the Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies* (*UBICOMM*), Florence, 247-256.

[64] Criado, N., Argente, E., Garrido, A., Gimeno, J.A., Igual, F., Botti, V., Noriega, P. and Giret, A. (2011) Norm Enforceability in Electronic Institutions? Coordination, Organizations, Institutions, and Norms in Agent Systems VI, Springer, 250-267. https://doi.org/10.1007/978-3-642-21268-0_14

[65] Florio, V.D. and Blondia, C. (2008) A Survey of Linguistic Structures for Application-Level Fault Tolerance. *ACM Computing Surveys*, **40**, 1-27.

[66] Weyns, D., Omicini, A. and Odell, J. (2007) Environment as a First Class Abstraction in Multiagent Systems. *Journal on Autonomous Agents and Multiagent Systems*, **14**.

[67] Paes, R., Lucena, C., Carvalho, G. and Cowan, D. (2009) An Event-Driven High Level Model for the Specification of Laws in open multi-agent systems. *Journal of Systems and Software*, **82**, 629-642. https://doi.org/10.1016/j.jss.2008.08.033

[68] Hu, J., Sun, Q.Z. and Chen, H.P. (2010) Application of Single Sign-On (SSO) in Digital Campus. 2010 3*rd IEEE International Conference on Broadband Network and Multimedia Technology* (*IC-BNMT*), IEEE, 725-727.

[69] Lynch, N.A. (1996) Distributed Algorithms. Morgan Kaufmann, San Francisco.

[70] Serban, C. and Minsky, N. (2009) *In Vivo* Evolution of Policies that Govern a Distributed System. *Proc. of the IEEE International Symposium on Policies for Distributed Systems and Networks*, London.