ADVANCES IN DECENTRALIZED AND STATEFUL ACCESS CONTROL

BY CONSTANTIN SERBAN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Naftaly H. Minsky

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2008

ABSTRACT OF THE DISSERTATION

Advances in Decentralized and Stateful Access Control

by CONSTANTIN SERBAN

Dissertation Director: Naftaly H. Minsky

The economy and security of modern society relies on increasingly distributed infrastructures and institutions, such as the banking system, government agencies, and commercial enterprises. This trend raises both the importance of access control technology and its complexity. Law-Governed Interaction (LGI) represents an advanced access control mechanism that satisfies many of the challenges posed by modern computing. LGI, however, has been defined for asynchronous, message passing, communication, leaving unsupported the wide range of applications that employ synchronous communication. Furthermore, no formal mechanism had been designed for adapting its policies in the presence of ever-changing security requirements. My dissertation addresses these issues as follows. It introduces Regulated Synchronous Communication, a novel access control model for synchronous, request-reply communication; it proposes Hot Updates, a mechanism for changing the policy of a distributed system while the system continues to operate.

Regulated Synchronous Communication extends the LGI mechanism to synchronous communication, thus providing advanced control over this important and popular mode of communication. Among the novel characteristics of this model are: the control of both the request and the reply; regulated timeout capability provided to clients, in a manner that takes into account the concerns of their server; and enforcement on both the client and server sides.

Hot Updates addresses the issue of changing the access control policy of a large distributed

system, in the context of LGI. Hot Policy Updates undertakes a number of challenges such as a) how to propagate the policy updates throughout the system, b) when to update the policy with respect to an individual component, and c) how to avoid, minimize or compensate possible inconsistencies that appear during the update process.

Both Regulated Synchronous Communication and Hot Updates had been implemented using Java Laws, a novel Java-based language for expressing access control policies for LGI. Java Laws provides a common platform for applying fine-grained access control particularly suitable for distributed applications written in Java. Among other advantages, Java Laws enables an efficient enforcement of access control, as well as good scalability and portability across various operating systems.

Acknowledgements

First and foremost I would like to thank Naftaly Minsky, my thesis advisor, for his constant guidance during the past eight years at Rutgers. His energy, drive, and unequivocal encouragement to perform research that matters was an inspiration.

I would like to thank Liviu Iftode for his contagious enthusiasm in advancing the research in the systems area; Ricardo Bianchini for his pragmatic approach, and selfless advice whenever I sought it; and Angelos Keromytis for his support as member of my thesis committee.

Many people have participated in the research in the area of Law-Governed Interaction over the time. I am grateful to all of them for adding their own contributions and for enhancing my understanding of what access control really is. I would like to extend special thanks to Wenxuan (Bill) Zhang for his help with the implementation and testing of the Java Laws and the controller, as well as for his performance measurements that I used in this dissertation. I would also like to thank Yukun Gou for his contribution to the design and initial implementation of the Hot Updates. I thank Takahiro Murata for the many insightful discussions that helped me understand what had already been discovered and what might be worth getting discovered; Victoria Ungureanu, Xuhui Ao, and Mihail Ionescu, who paved my way in many respects.

I would like to thank Cristian Borcea for his constant support in school and afterschool, for so many years that neither of us is willing to acknowledge. It is due to him that I ever started to think about graduate studies.

Most of all, I thank my parents, that gave me both the nature and the nurture to take me through the perils of grad school; my son, Alex, who slept when I needed not; and my wife, Anya, who lost her sleep for me to have it.

Dedication

To My Family

Table of Contents

List of Figures

List of Abbreviations

AC	Access Control
CS	Control State
LGI	Law-Governed Interaction
RMI	Remote Method Invocation
RRMI	Regulated Remote Method Invocation
RPC	Remote Procedure Call
VM	Virtual Machine
XACML	eXtensible Access Control Markup Language

Chapter 1

Introduction

The economy and security of modern society relies on increasingly distributed infrastructures and institutions—such as the power grid, the banking system, transportation, medical institutions, government agencies, and commercial enterprises. This trend increases both the importance of *access control* (AC) technology and its complexity. The importance of access control is increased because such critical systems often communicate via the Internet and can no longer protect themselves by hiding within their local intranet behind their firewalls [11]. Rather, they now depend on access control to protect them against malicious attacks by regulating the messages exchanged among their users or components, and between such systems and the outside world. While the conventional access control mechanisms are still largely centralized and based on the access control matrix model, often upgraded into "role-based AC" (RBAC) [57] [49], the limitations of these mechanisms have been long recognized in the context of commercial [19] and clinical [3] applications. The shortcomings of conventional models become apparent when the complexity of the application domain and the requirements facing access control are taken into account. Accordingly, access control mechanisms need to exhibit a number of characteristics that enable them to successfully address the security of modern systems, such as: (a) support for *expressive* policies; (b) strict control of the interaction occurring in large and distributed communities of agents, via *communal* (overarching) policies; (c) a *scalable* and *decentralized* enforcement mechanism that is able to cope with large-scale systems, as employed in federations of enterprises or grid computing.

1.1 The Characteristics of Modern Access Control

The expressiveness of a policy reflects its capacity to make decisions and take actions in an wide array of circumstances. An important attribute of expressiveness is the sensitivity to the

history of interaction, which defines the so-called *stateful*, or *dynamic* policies. One's ability to perform specific actions depends on a certain dynamic state maintained by the access control mechanism, representing such properties as acquired roles, credentials, or attributes [4]. This state, which we call control state, in turn depends on previous actions taken by the agent, thus reflecting a history of interaction relevant to the policy at hand [34] [22]. The stateful character of a policy is critical in financial systems, where interactions often occur according to budgetary constraints, but it is important in other kinds of systems as well. Other types of stateful policies include, in particular, dynamic separation of duties [27][58][36] and Chinese-Wall policies [15] [59]. Another attribute of expressiveness is the degree of *initiative* manifested by the policy. Conventional AC policies are limited to permitting or prohibiting messages. But one often needs to take other actions when observing the sending or the receipt of a message, such as sending a copy of the message to some audit trail server, triggering a delayed action, or changing the state of the policy, in the case of stateful policies. Some of these capabilities have been introduced into several recent AC models. In particular, the AC model of Ryutov and Neuman [56] supports policies that can exhibit simple initiatives, but they do not support stateful policies; the same is true for XACML [29], a recent AC standard for web-services.

Communality represents another characteristic of access control, prevalent in large distributed enterprise systems. Most conventional AC mechanisms are designed for *server-centric* policies. Such policies are employed by individual servers in order to regulate the access to their own resources, and are usually expressed via Access Control Lists, or via a formalism like the Keynote [14]. The enforcement mechanism for server-centric policies consists of a reference-monitor that mediates the interactions of the server with its clients. This reference monitor is usually run by the server itself, or is closely associated with it. But the server-centric approach is inadequate for the growing class of applications where the interactions among the members of a distributed community of servers and clients—or a community of peers—is subject to an overarching *communal* policy. In such a community, the particular interaction between a client a server affects the client's ability to get services from any server in the AC domain. Thus, communal policies contain aggregate sets of rules that control the interaction between multiple servers and their clients, in a homogeneous and unitary manner. The importance of communal, enterprise-wide policies has been recently recognized by some academic projects [26], as well

as by commercial systems such as IBM-Tivoli [38], and by XACML [29].

Decentralized enforcement reflects the degree of distribution employed by an access control enforcement mechanism. Most enterprise-wide mechanisms for access control employ a centralized reference monitor to mediate all interaction between agents in the enterprise, subject to a given communal policy. This reference monitor is often replicated, for the sake of scalability. But none of these mechanisms and models support fully stateful policies—and for a good reason. As argued in [5], it is hard to scale global stateful policies through the use of standard replication techniques because a state change sensed by one replica of the reference monitor may have to be propagated atomically to all other replicas. As a consequence, for an AC mechanism to support communal and stateful policies in a scalable manner, it needs to be *decentralized*. This decentralization applies to both the maintenance and evaluation of the communal access control policy, as well as to the maintenance of the control state.

Previous research on distributed access control has shown that the above needs can be addressed successfully by Law-Governed Interaction (LGI) [47, 5, 6]. LGI is a message-exchange mechanism that allows an open and heterogeneous group of distributed actors to engage in a mode of interaction governed by an explicitly specified and strictly enforced policy, called the law of this group. Due to its flexibility and expressivity, LGI is a generalization of the conventional concept of access-control. It also represents a radical departure from conventional AC mechanisms in that it employs an inherently decentralized policy-enforcement technique.

Access control mechanisms, however, are often dependent on the communication models and protocols supporting the interaction within a system. LGI has been defined so far for asynchronous (message passing) communication, leaving unsupported the wide range of applications that employ synchronous communication—by which we mean here a request-reply type of interaction, when the client thread is blocked while waiting for the reply[1]. The access control for synchronous communication, however, has its own specific requirements, which are different from those of asynchronous one, particularly when dealing with communal and

[1]The term “synchronous communication” as used here is not to be confused with the notion of “synchronous send”, which requires the sender to wait for an acknowledgment of receivership before proceeding further in its computation; our definition assumes an exchange of payload information both at the request and at the reply time. Among the communication protocols supporting this type of synchronous communication are SunRPC, JAX-RPC, CORBA, DCOM, and Java RMI.

stateful policies. Previous research has not addressed the requirements and the effects of synchronism on access control. Another important aspect of an access control mechanism that has not been addressed so far, is the ability to update the access control policies of a system while the system continues to operate. Such updates are particularly challenging when the policies themselves are distributed, as in the case of LGI, and when the impact of the update on the system is to be minimized.

1.2 Dissertation Contributions

My thesis introduces **Regulated Synchronous Communication**, an advanced access control mechanism that proposes the control of the reply and of the timeout of synchronous communication. It furthermore presents **Hot Updates**, a novel mechanism for propagating policies throughout a widely distributed system, without incurring inconsistencies. It also introduces **Java Laws**, a Java-based language for expressing fine-grained access control policies particularly suitable for distributed applications written in Java.

A number of characteristics of synchronous communication can have potential impact on access control: a) the bi-directional aspect of communication, with payload information transfered both from client to server and from server to client, and b) the timing of the interaction, manifested as a blocking time at the client side, or as a time-out notification on the server side. These aspects of synchronous communication had not been previously taken into account when designing access control mechanisms. Regulated Synchronous Communication represents an advanced access control mechanism that exhibits a number of novel characteristics such as: (a) the control of both the request and the reply parts of a call, separately, but in a coordinated fashion; and (b) a regulated timeout capability, taking into account the concerns of both the server and the client. The implementation of this model has given rise to a specific communication protocol, called Regulated Remote Method Invocation, or RRMI, representing a versatile, security-enabled version of Java RMI [71].

One of the defining characteristics of a policy-based system in general, and of a policy-based access-control ensemble in particular, is the separation of the policy from the mechanism [73]. This separation enables different policies to apply to the system without changing its

underlying mechanism, thus yielding a more robust system, able to adapt to various security requirements. The ability to seamlessly change the policy is one of the most salient features, and it constitutes at the same time an integral part of any policy-based system. The process of changing the policy while the system continues to operate, is called a hot update. The update process can vary widely with the type of policy and system in question. In traditional access control mechanisms the policy can be changed atomically by suspending the system, replacing the policy, and subsequently resuming the activities. The updating process, however, becomes more challenging for applications with critical availability requirements. The problem is furthermore compounded in the case of decentralized systems, where the policy and its specific control state are distributed on a large scale, as in the case of LGI. In such systems, an atomic update becomes too disruptive; an incremental update can be employed instead, such that the policy is changed individually for all the components.

Hot Updates represents a flexible model for updating the policies of LGI. It addresses a number of issues, such as: a) how to propagate the policy updates throughout the system, b) when to update the policy with respect to an individual component, and c) how to avoid, minimize or compensate possible inconsistencies that appear during the update process. The model introduces a mechanism for promoting the updates at both individual, i.e., component level, and at a system level, as well as support for resolving inconsistencies that appear when different components are simultaneously subject to different versions of an access control policy. Hot Updates maintains flexibility by providing various methods for propagating the updates, suitable for different systems subject to a wide range of access control policies.

An important goal of this dissertation was to provide an access control mechanism that manifests high efficiency when applied to various classes of applications, deployed in different environments. Java Laws represents a novel Java-based language for expressing access control policies designed to achieve this goal, for LGI in general, and for Regulated Synchronous Communication and Hot Updates in particular.

In LGI, Java Laws allows for an efficient evaluation of policies by integrating the policy evaluation module with the LGI-specific reference monitor, called controller, itself a Java component. This integration offers portability by enabling the deployment of the enforcement infrastructure across various operating systems and platforms. Java Laws also provides a common

platform for applying fine-grained access control particularly suitable for distributed applications written in Java. For such applications, Java Laws provides the mechanism to interpret the traffic at multiple levels, both as binary data, or, at a higher level, as formatted Java objects, thus leveraging the understanding of the occurring interaction, necessary for sophisticated access control decision. This feature becomes particularly useful in the context of Regulated Synchronous Communication, whose implementation—Regulated RMI—represents a variant of the popular Java RMI protocol. In this context, the data exchange represents Java objects exclusively. A complete comprehension of these objects, using Java specific methods, allows us to exert a fine-grain control over the application at hand. We have evaluated Java Laws in a number of experiments, in order to assess its efficiency. Our results indicate that Java Laws introduces a relatively small overhead in the communication between plain Java applications, and compares favorably with similar security-enabled Java-based communication protocols.

1.3 The Road to Dissertation

My passion for distributed systems and security dates back at least a decade, predating my graduate studies at Rutgers. It was at Rutgers, however,—after joining the Security and E-Commerce Lab—that I really understood what are the complexities and challenges facing access control in large scale distributed systems. This dissertation contains only the highlights of my work after starting my research on Law-Governed Interaction. During this period, I pursued several other projects that either laid down the foundation of my thesis, or explored related venues. Even though these projects did not make it through this document, they represent valuable contributions.

One of the first projects that I participated to was the “Secretary” project, an admission control mechanism for distributed communities [65]. The “Secretary” enabled the establishment of explicit groups of agents operating within indefinite, open, and volatile communities, while providing membership and long-term storage services. The project has been later extended for providing naming services, aliasing, and anonymity for explicit groups [74].

In parallel with my work at the “Secretary” project, I started to work on Java Laws, motivated by the need to have a more performant, reliable, and portable language for interpreting

laws. The first workable version of Java Laws was quickly followed by the implementation of the Regulated RMI suite [68, 67]. The performance of the Java Laws, of the Regulated RMI, as well as of the LGI in general, had been furthermore boosted after I have redesigned and implemented the LGI middleware from the ground-up. The middleware consists of a Java package containing the controller and a number of other tools supporting LGI, such as a controller manager, a law server, as well as the Java and Prolog based law interpretors. The middleware provides sophisticated solutions to communication security (public key, certificates, secure hashing), an enhanced GUI (user interface applets, swing interfaces and HTTP rendering), a suite of HTTP and TCP/IP servers, and Java-to-Prolog interfaces that enhance the portability of Prolog laws.

Following this implementation, the middleware became mature enough and had been afterwards released for public use. As part of this momentous release effort, I created and am currently maintaining its website [63]; I prepared a number of online tutorials, an example suite; and, together with Naftaly Minsky, I co-authored the LGI Reference Manual [45].

As a benefit of the effort of enhancing the efficiency, portability, and manageability of the LGI middleware, I could undertake—together with my colleagues in the Security and E-Commerce Lab—a series of projects designed to demonstrate the use of LGI in a number of specific areas. The most prominent of these project was the implementation of a decentralized and secure marketplace [66, 18, 17], a concept designed to bring trust between the buyers and sellers of a virtual marketplace, where such trust cannot be achieved by geographical proximity, societal or governmental laws, or by implicit rules embedded in a centralized server.

Finally, in a somewhat different direction, I studied the enforcement of interaction properties on homogeneous and centralized systems using Aspect-Oriented Programming techniques [39]. As part of this project, I studied the negative impact of the interference between security aspects and development aspects which might coexist in an application [69]. Furthermore, I have developed a specialized compiler, called AspectJTamer [70], intended to aid the builders and integrators of applications based on AspectJ [9]. AspectJTamer provides support for identifying aspects that are present in binary components as well as for determining their characteristics. Additionally, AspectJTamer provides a mechanism to control the scope of binary aspects on a per-class granularity, using controlled weaving and extraction directives.

1.4 Dissertation Plan

This dissertation is organized as follows. Chapter 2 provides an overview of LGI, and describes its most important features. In Chapter 3 we describe Regulated Synchronous Communication, an extension of LGI designed to provide advanced access control for synchronous communication, along with its RMI-specific implementation. Chapter 4 presents Hot Updates, the mechanism that supports the update of policies for a distributed system, together with a number of methods used for policy dissemination. Java Laws, the Java-based language used for expressing access control policies in Regulated Synchronous Communication and Hot Updates, is presented in Chapter 5. Chapter 6 presents future work, and the dissertation concludes with Chapter 7.

Chapter 2

An Overview of LGI

In this chapter, we provide an overview of the Law-Governed Interaction (LGI). We start with a description of the concept of law and its local nature, and we present the law enforcement mechanism and the infrastructure that allows for a distributed and scalable deployment. We continue with a description of the most original aspects of LGI: we present the use of digital certificates for establishing trust among different components of the system, and we describe the obligation and exception mechanisms used for enhancing the capabilities of the law. We conclude the chapter with a discussion about the maintenance of the infrastructure of LGI.

2.1 The Concept of LGI

LGI is a mode of interaction that allows an open group of distributed heterogeneous *agents* to interact with each other with confidence that the explicitly specified policies, called the *law* of the open group, is complied with by everyone in the group [47, 43]. The messages exchanged under a given law $\mathcal{L}$ are called $\mathcal{L}$-$messages$, and the group of agents interacting via $\mathcal{L}$-messages is called a *community* $\mathcal{C}$, or more specifically, an $\mathcal{L}$-$community$ $\mathcal{C}_{\mathcal{L}}$.

The concept of ”open group” has the following semantic: (a) the membership of this group can be very large, and can change dynamically; and (b) the members of a given community can be heterogeneous. Such open groups are often encountered in business applications, as advocated by service-oriented architectures [51] [53, 21] [35]. LGI does not assume any knowledge about the structure and behavior of the members of a given $\mathcal{L}$-community. All such members are treated as black boxes by LGI. LGI only deals with the interaction between these agents. Members of a community are not prohibited from non-LGI communication across the Internet, or from participation in other LGI-communities.

For each agent x in a given $\mathcal{L}$-community, LGI maintains the control state $\mathcal{CS}_x$ of this

Main regulated events	
`sent(x,m,y)`	takes place at x when x sends a message to y;
`arrived(x,m,y)`	takes place at y when a message from x arrives;

Other regulated events	
`adopted(x,[a])`	is the event that marks the fact that x started to operate under this law;
`certified(x,cert(I,S,A))`	is associated with the submission of a certificate. I is the issuer (the Certifying Authority); S is the subject; A are the attributes of the certificate.

Figure 2.1: Regulated events in LGI

agent. These control states, which can change dynamically, subject to law $\mathcal{L}$, enable the law to make distinctions between agents, and to be sensitive to dynamic changes in their states. The semantics of the control state for a given community is defined by its law, and could represent such things as the role of an agent in this community, its privileges and reputation. The $\mathcal{CS}_x$ is a bag of objects called *Terms*. For instance, a Term with the value `role(manager)` in the control state of an agent might denote that the agent has been authenticated to be a manager of a given organization. The middleware implementing LGI , its supporting documentation, and an online infrastructure for public access are available for free on its website at [63].

2.2 The Law and Its Enforcement

Generally speaking, the law of a community $\mathcal{C}$ is defined over certain types of events occurring at members of $\mathcal{C}$, mandating the effect that any such event should have; this mandate is called the ruling of the law for a given event. The events subject to laws, called *regulated events* include, among others: the sending and the arrival of an $\mathcal{L}$-message; the coming due of an obligation previously imposed on a given object; and the submission of a digital certificate. A number of primitive operations is presented in Figure 2.1. The operations that can be included in the ruling of the law for a given regulated event are called *primitive operations*. They include: operations on the control state of the agent where the event occurred (called, the ”home agent”); operations on messages, such as forward and deliver; and the imposition of an obligation on

Operations on the control-state	
`doAdd(t)`	adds term `t` to the control state;
`doRemove(t)`	removes term `t` from the control state;
`doReplace(t1 t2)`	replaces term `t1` with term `t2`;
`incr(t(v),d)`	increments the value of the parameter `v` of term `t` by quantity `d`
`dcr(t(v),d)`	decrements the value of the parameter `v` of term `t` by quantity `d`

Operations on messages	
`forward(x,m,y)`	sends message `m` from `x` to `y`
`deliver(x,m,y)`	delivers the message `m` from `x` to agent `y`

Figure 2.2: Primitive operations in LGI

the home agent. A sample of primitive operations is presented in Figure 2.2.

Note that the ruling of the law is not limited to accepting or rejecting a message, but can mandate any number of operations, like the modifications of existing messages, and the initiation of new messages and of new events, thus providing the laws with a strong degree of flexibility. More concretely, LGI laws are formulated using an event-condition-action pattern. Throughout this thesis we will depict a law using the following pseudo-code notation:

`upon` $\langle event \rangle$ `if` $\langle condition \rangle$ `do` $\langle action \rangle$

Where the $\langle event \rangle$ represents one of the regulated events, the $\langle condition \rangle$ is a general expression formulated on the event and control state, and the $\langle action \rangle$ is one or more operations mandated by the law. This definition of the law is abstract in that it is independent of the language used for specifying laws. The language used initially for expressing laws in LGI was Prolog. Chapter 5 will introduce a Java-based language and will present its advantages. But despite the pragmatic importance of a particular language being used for specifying laws, the semantics of LGI is basically independent of that language.

A law $\mathcal{L}$ can regulate the exchange of messages between members of an $\mathcal{L}$-community, based on the control state of the participants; and it can mandate various side effects of the message exchange, such as modification of the control states of the sender and/or receiver of a message, and emission of extra messages.

2.2.1 The Local Nature of Laws

Although the law $\mathcal{L}$ of a community $\mathcal{C}$ is global in that it governs the interaction between all members of $\mathcal{C}$, it is enforced locally at each member of $\mathcal{C}$. This is accomplished by the following properties of LGI laws:

- $\mathcal{L}$ only regulates local events at individual agents.
- The ruling of $\mathcal{L}$ for an event e at agent x depends only on e and the local control state $\mathcal{CS}_x$ of x.

The ruling of $\mathcal{L}$ at x can mandate only local operations to be carried out at x, such as an update of $\mathcal{CS}_x$, the forwarding of a message from x to some other agent y, and the imposition of an obligation on x. The fact that the same law is enforced at all agents of a community gives LGI its necessary global scope, establishing a common set of ground rules for the members of C and providing them with the ability to trust each other, in spite of the heterogeneity of the community. Furthermore, the locality of law enforcement enables LGI to scale with the size of the community.

2.2.2 Distributed Law-Enforcement

Broadly speaking, the law $\mathcal{L}$ of community $\mathcal{C}_\mathcal{L}$ is enforced by a set of trusted agents, called *controllers*, that mediate the exchange of $\mathcal{L}$-messages between members of $\mathcal{C}_\mathcal{L}$. Every member x of $\mathcal{C}$ has a controller T_x assigned to it (T here stands for trusted agent) which maintains the control state $\mathcal{CS}_x$ of its client x. All these controllers, which are logically placed between the members of $\mathcal{C}$ and the communication medium as illustrated in Figure 2.3 carry the same law $\mathcal{L}$. Every exchange between a pair of agents x and y is thus mediated by their controllers T_x and T_y, so that this enforcement is inherently decentralized. However, several agents can share a single controller, if such sharing is desired. The efficiency of this mechanism, and its scalability, are discussed in [47].

Controllers are generic, and can interpret and enforce any well-formed law. A controller operates as an independent process, and it may be placed on any machine, anywhere in the network. We have implemented a controller-service, which maintains a set of active controllers.

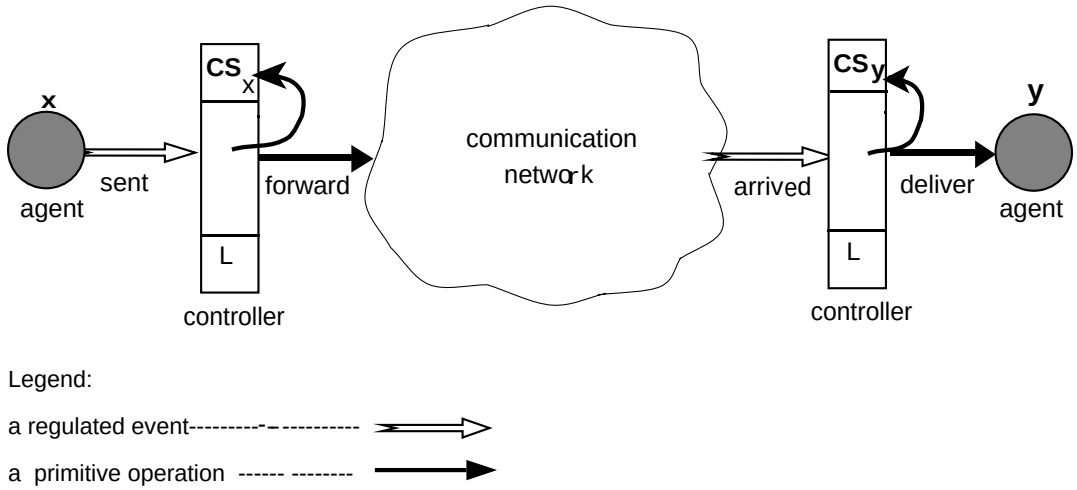

Figure 2.3: Enforcement of the law

To be effective in a widely distributed enterprise, this set of controllers need to be well dispersed geographically, so that it would be possible to find controllers that are reasonably close to their prospective clients.

2.2.3 The basis of trust between members of a community

For members of an $\mathcal{L}$-community to trust its interlocutors to observe the same law, one needs the following assurances: (a) Messages are securely transmitted over the network; (b) The exchange of $\mathcal{L}$-messages is mediated by controllers interpreting the same law $\mathcal{L}$; and (c) All these controllers are correctly implemented. If these conditions are satisfied, then it follows that if agent y receives an $\mathcal{L}$-message from agent x, this message must have been sent as an $\mathcal{L}$-message; in other words, that $\mathcal{L}$-messages cannot be forged.

Secure transmission is carried out via traditional cryptographic techniques. To ensure that a message forwarded by a controller T_x under law $\mathcal{L}$ would be handled by another controller T_y operating under the same law, T_x appends the one-way hash [61] H of law $\mathcal{L}$ to the message it forwards to T_y. T_y would accept this as a valid $\mathcal{L}$-message if and only if H is identical to the hash of its own law.

As to the correctness of controllers, we assume here that every $\mathcal{L}$-community is willing to trust the controllers certified by a given certification authority (CA), which is specified by the law $\mathcal{L}$. In addition, every pair of interacting controllers must first authenticate each other

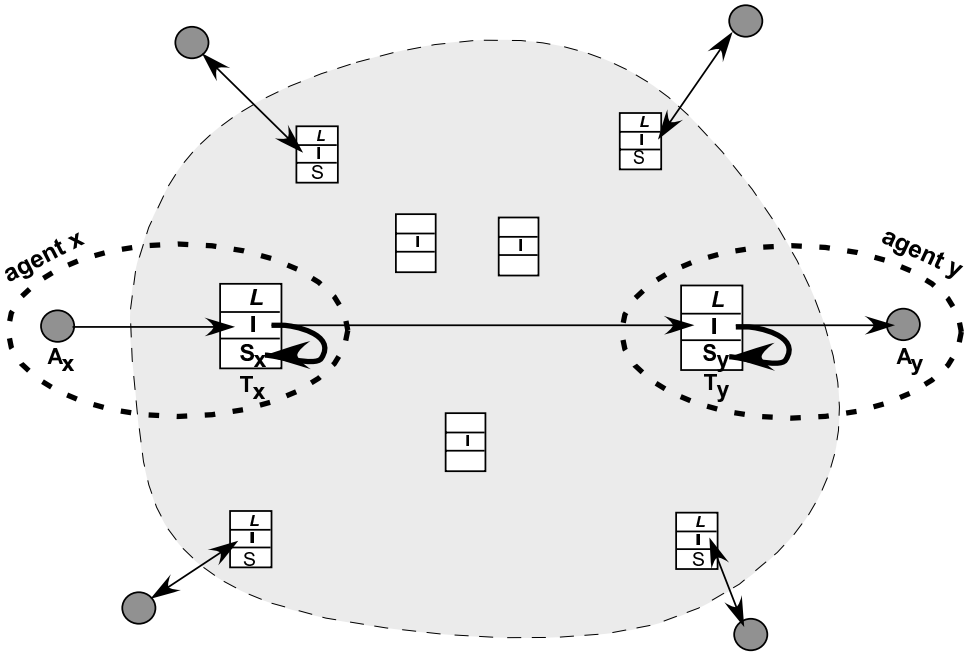

Figure 2.4: LGI controller infrastructure

by means of certificates signed by this CA. This requires the existence of a trusted set of controllers, maintained by what we call a controller-service, or CoS, to be discussed below.

2.2.4 Engaging in an $\mathcal{L}$-Community

For an agent x to be able to exchange $\mathcal{L}$-messages with other members of an $\mathcal{L}$-community, it must: (a) find an LGI controller, and (b) notify this controller that it wants to use it, under law $\mathcal{L}$. As mentioned before, a controller can be located by contacting a *controller-service*, which represents a set of active controllers that are available for intermediating the interaction with a given agent. The controllers can be dispersed geographically over the Internet or distributed enterprise, so one agent can select a controller reasonably close to it. Figure 2.4 displays such a controller service and shows how agents are associated with individual controllers. Actors are depicted by circles, controllers are represented as boxes operating under law L. Agents are depicted by dashed ovals that enclose (actor, controller) pairs. Thin arrows represent messages, and thick arrows represent modification of state.

Upon selecting a controller T, x would contact it by providing two parameters: a `law` and

an `id`. The `law` parameter represents the law x wants to operate under, and `id` is the name that it wants to be known by within this community. Upon receiving such a request, the controller T checks the supplied law for syntactic validity, and the chosen id for uniqueness among the identifiers of all current agents handled by T. If these two conditions are satisfied, and if T is not already loaded to capacity, it will set up a control state structure for agent x, allowing it to start operating under this law[1].

2.3 Some Advanced Features of LGI

We introduce here briefly some of the advanced features of LGI, in particular those employed in this thesis. For additional information about these features, and for a study of their use, the reader is referred to the LGI manual [45].

2.3.1 The Treatment of Certificates

The conventional usage of certificates includes: authentication of the identity of an agent; authentication of the role a given agent plays in a certain community; and testimonial of certain rights that a given agent obtained from another via delegation [32], [33], [10].

Certificates may be required by a given law $\mathcal{L}$ to certify the controllers used to interpret this law. Certificates may also be submitted by an actor Ax to its controller Tx. The effect of such certificates is subject to the law in question. Typically, such submitted certificates are used to authenticate the identity of the actor, or the role it plays in the environment in which the community in question operates [7].

For now, the middleware implementing LGI supports SPKI/SDSI model [25] for certificates. But it would be very easy to adapt LGI to any other structure one may prefer. Under LGI, a certificate is a four-tuple (issuer, subject, attributes, signature), where issuer is the public key of the CA that issued and signed this certificate, subject is the public key of the principal that is the subject of this certificate, attributes is what is being certified about the subject, and

[1] If any one of these conditions is not satisfied, then x would receive an appropriate diagnostic, and will be able to try again.

the signature is the digital signature of this certificate by the issuer. The attributes field is essentially a list of (attribute, value) pairs. For example, the attributes of a certificate might be the list [name(johnDoe), role(employee)], asserting that the name of the subject in question is John Doe and its role in this enterprise is an employee.

2.3.2 Enforced Obligation

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

Specifically, an obligation can be imposed on a given agent x at time t_0 by the execution at x of a primitive operation `imposeObligation(oType,dt)`, where `dt` is the time period, after which the obligation is to come due, and `oType`—the obligation type—is a term that identifies this obligation (not necessarily in a unique way). The main effect of this operation is that unless the specified obligation is *repealed* before its due time $t = t_0 + dt$, the regulated event `obligationDue(oType)` would occur at agent x at time t. The occurrence of this event would cause the controller to carry out the ruling of the law for this event; this ruling is, thus, the sanction for this obligation. Note that a pending obligation incurred by agent x can be repealed before its due time by means of the primitive operation `repealObligation(oType)` carried out at x, as part of a ruling of some event. (This operation actually repeals *all* pending obligations of type `oType`.)

2.3.3 Interoperability Between Communities

LGI also supports the interoperability between different communities. By interoperability we mean, the ability of an agent `x` operating under law $\mathcal{L}_x$ to exchange messages with agent `y` operating under different law $\mathcal{L}_y$, such that the following properties are satisfied: (a)`consensus:`

An exchange between a pair of laws is possible only if it is authorized by both laws. (b)`autonomy`: The effect that an exchange initiated by `x` operating under law $\mathcal{L}_x$ may have on the structure and behavior of `y` operating under law $\mathcal{L}_y$, is subject to law $\mathcal{L}_y$. (c)`transparency`: Interoperating parties need not to be aware of the details of each other's law.

To support such an interoperability between communities, LGI uses slightly different primitive operations and events than those used for communication within the same community:

- `forward(x,m,[y,`L_y`])`: invoked by agent `x` under law $\mathcal{L}_x$, initiates an exchange between `x` and agent `y` operating under law $\mathcal{L}_y$. When the message carrying this exchange arrives at `y` it would invoke at it an `arrived` event under $\mathcal{L}_y$.
- `arrived([x,`L_x`],m,y)`: occurs when a message `m` exported by `x` under law $\mathcal{L}_x$ arrives at agent `y` operating under law $\mathcal{L}_y$.

Exactly what laws one can interoperate with is defined by a `Portal` clause in the preamble of each law, thus there is a precise definition of each such exchange between communities.

2.3.4 The Treatment of Exceptions

Primitive operations that initiate messages, like *deliver* and *forward*, may end up not being able to fulfill their intended function. For example, the destination agent of a forward operation may fail by the time the forwarded message arrives at it. Such failures can be detected and handled via a regulated event called an *exception*, which is triggered when a primitive operation that initiates communication cannot be completed successfully. It is up to the law to prescribe what should be done to recover from such an exception. The syntax of an exception event is: `exception(op, diagnostic)`, where op is the primitive operation that could not be completed, and diagnostic is a string describing the nature of the failure. The home of the exception event is the home of the event that attempted to carry out the failed operation. For instance, if a message m, forwarded by an agent x to an agent y operating under law $\mathcal{L}$ cannot reach its destination, then an event `exception(forward(x,m,[y,L]),``destination not responding'')` would be triggered at x. Commonly, exceptions are triggered by the forward and deliver primitive operation, as well as other communication primitives. More details about the exception mechanism are given in the LGI manual [45].

2.3.5 The Hierarchical Organization of Laws

LGI provides a mechanism to organize the laws into *hierarchies* [8, 6]. Each such hierarchy, or tree, of laws $t(\mathcal{L}_0)$, is rooted in some law $\mathcal{L}_0$. Each law in $t(\mathcal{L}_0)$ is said to be (transitively) subordinate to its parent, and (transitively) superior to its descendents. Generally speaking, each law $\mathcal{L}$' in a hierarchy $\mathcal{L}_0$ is created by refining a law $\mathcal{L}$, the parent of $\mathcal{L}$', via a $\Delta\mathcal{L}$', where a Δ is a collection of rules defined as a refinement of an existing law. The root $\mathcal{L}_0$ of a hierarchy is a normal LGI law, except that it is created to be open for refinements, using a consulting function. This function allows the root law to suggest (pseudo) events to its subordinate Δ, and to receive, and possibly interpret a proposed ruling. The final decision about the ruling of law $\mathcal{L}$' is made by its superior law $\mathcal{L}$, leaving its Δ only an advisory role. Thus, the process of refinement is defined in a manner that guarantees that every law in a hierarchy conforms (transitively) to its superior law.

2.4 The Controller Infrastructure

For LGI to be scalable enough to support a large and geographically distributed community, it needs to employ a reliable and secure set of controllers, which collectively constitute the trusted computing base (TCB) of LGI. Such an infrastructure of controllers is called the controller service, or CoS.

For use within an enterprise, such a CoS can be maintained and managed by the enterprise administration, and can thus be trusted by all enterprise computations. But for the CoS to support a truly open community, to be used by people and servers distributed all over the Internet, and not belonging to any single administrative domain, the CoS needs to function as a public utility. There are no serious technical impediments to the construction of a CoS public service. But it needs to be done by a large financial or governmental organization that can serve as a trusted third party, with no financial interest in the computing activities regulated by its controllers. This organization must assume certain liabilities for various failures of the controllers provided to its customers. It also needs to provide audit trail of its controllers' activities, which are secure enough to be accepted in a court of law, in case of a dispute.

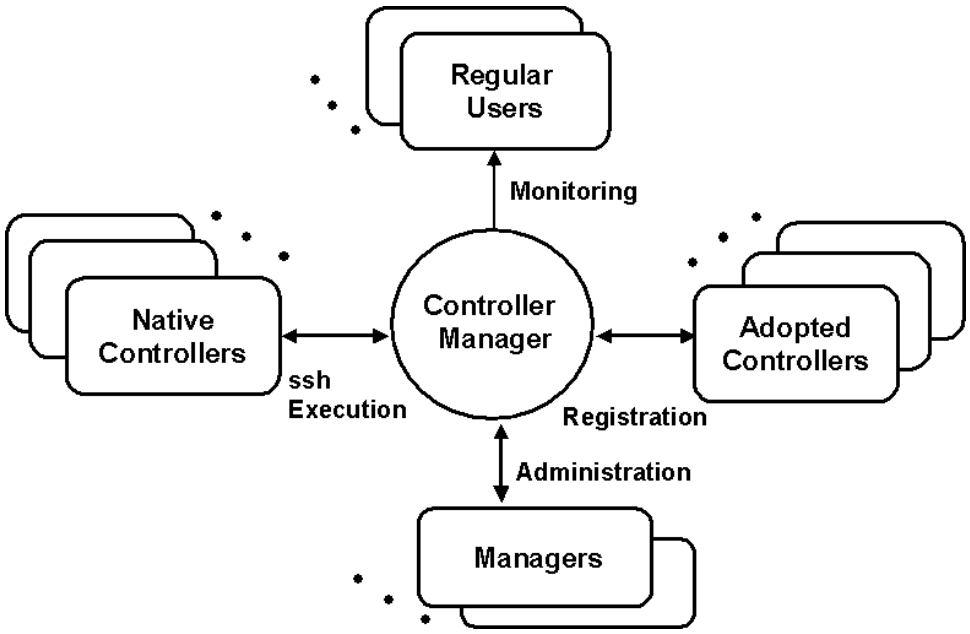

Figure 2.5: The interaction with the Controller Manager

2.4.1 The Controller Manager

Given the importance of such a trusted infrastructure of controllers for the functionality of LGI, we have designed and constructed a tool, called *controller manager*, responsible for the management of such an infrastructure. The controller manager serves a double purpose:

- *name server*: lists a number of available controllers; provides lookup services to prospective agents, who want to operate under LGI.
- *manager*: maintains a controller service infrastructure ; it helps to start, monitor, and stop the controllers that make up such an infrastructure.

Figure 2.5 presents the interaction model of the controller manager. The interacting entities can be classified into two categories: users and controllers. The controller manager distinguishes between regular users (human or software) and administrators. Regular users employ the lookup capabilities of the controller manager, while administrators manage the controller service. The controller manager can create, test, and destroy controllers on behalf of the administrators using a secure shell communication to a cluster of available infrastructure machines.

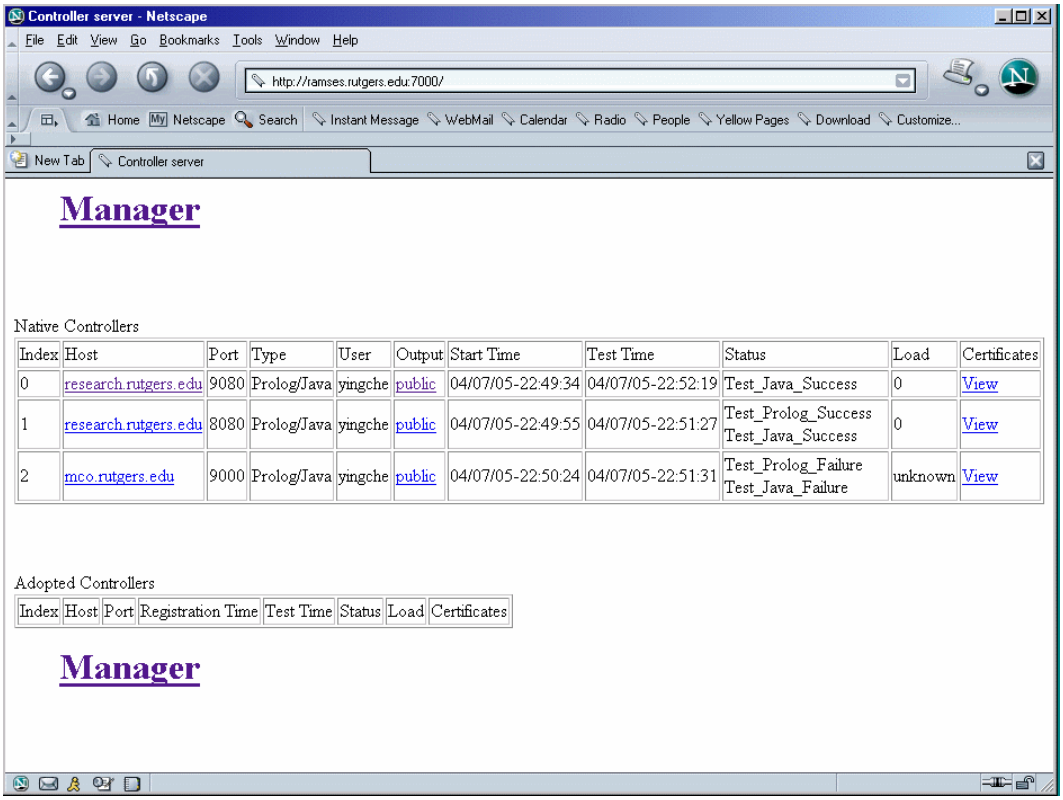

Figure 2.6: User view - controller lookup

Additional controllers that are not managed directly by the controller manager can register with the controller manager. The manager performs testing and authentication on such controllers.

User Interface

The main function of the controller manager is to provide controller lookup information for all the users, either humans or software. Users can access the controller directory by consulting the Controller Manager using an http-based interface. Figure 2.6 presents a sample of the view provided by the controller manager. The information is displayed separately for managed controllers and for registered controllers. Among the information displayed are such things as: the address of the controller, the supported language for the laws, the usage, and the status.

Administrative Interface

Administrators can connect to the controller manager and perform managing jobs through a specific administrative interface. There are four types of activities a manager can perform: a) start/create a controller, b) stop/destroy a controller, c) test the status of a controller or of the entire controller service, d) configure the controller manager. While the starting and stopping of the controllers are on-demand activities, the testing can be performed both on demand, or automatically. The automatic procedure tests the entire infrastructure periodically by both querying the status of the controllers and by using a special testing law that validates the behavior of the controllers for specific scenarios.

2.5 Summary

In this chapter, we have provided an overview of LGI. LGI represents a decentralized coordination and control mechanism for distributed systems. At the core of LGI is the concept of law, representing an explicitly stated and strictly enforced set of rules governing the behavior of each agent. The law is enforced by a set of generic components, called controllers. The most prominent features of LGI are: its support for sophisticated and powerful policies; stateful character, due to maintaining a control state on behalf of each agent; and inherent scalability enabled by a local formulation of the laws.

Chapter 3

Regulated Synchronous Communication

This chapter describes Regulated Synchronous Communication, a generalized access control mechanism for synchronous communication. We start with a case study illustrating a typical access control scenario in a distributed system. We then motivate the need for a specific access control method for synchronous communication. In Section 3.3, we describe the architecture of our proposed mechanism, and we show how it supports the policy introduced in the case study. We continue by presenting the implementation of this mechanism for the Java RMI protocol, giving rise to what we call Regulated RMI (or RRMI). We conclude the chapter after a brief discussion of related work.

3.1 A Pay-Per-Service Interaction: a Case Study

In order to illustrate the specific aspects of access control entailed by synchronous communication, let us begin with a simple case study. Consider a large, geographically distributed hospital whose management decided that all internal services—such as drug acquisition (from internal pharmacies), printing, file-services, record databases, etc.— would operate as *cost centers*. Accordingly, services need to be paid with internal currency, made available to various clients in their *e-wallets*. More specifically, the requests for such services and the budgeting of these requests are to be regulated by the following policy, to be called PPS for “pay-per-service”.

1. *An agent that plays the role of a* budget officer *can provide any amount of currency to any agent in the enterprise, to be maintained in the* e-wallet *of that agent.*

2. *Each service request must carry a payment, which is to be deducted from the e-wallet of the client. When the service has been carried out successfully, this payment is to be deposited in the e-wallet of the server. (A service is considered successful if it does not terminate with an exception.)*

3. *A client can cancel a service while it is being handled by the server, incurring a penalty that amounts to a fraction f of the price of a normally completed service. This penalty is to be payed to the server, while the rest of the original payment is to be returned to the client.*

Note that policies of this kind can be used for budgetary control of systems, whether or not the budget has any monetary connotation.

The pay-per-service policy represents a challenge to traditional access control, due to several characteristics. First, PPS is sensitive to the history of communication, i.e. stateful, since one's ability to make service requests depends on the amount of currency in its e-wallet, which, in turn, depends on previous service requests it has made; this state, containing the e-wallet of each agent, will be referred below as the control state. Second, PPS policy is clearly communal, in particular, because the content of the e-wallet of an agent effects the ability of that agent to get services from any server in the distributed hospital, or the AC domain.

Additionally, as we shall see, PPS places additional demands on access control when the distributed system employs synchronous communication.

3.2 Motivation

Synchronous communication differs from message passing in two respects: a) it consists of both a request and a reply, and payload is potentially exchanged in both steps of the interaction;

and b) it assumes a specific duration, representing the time the client is blocked waiting for the reply, or the time the server computes the results. In this section, we will argue that we need to: a) regulate both the request and the reply parts of a synchronous call, separate, but in a coordinated manner, and b) control the timing of the interaction.

3.2.1 The Need to Regulate Both the Request and the Reply Parts of a Call

Conventional access control mechanisms for synchronous communication regulate only the request step of a call, leaving the reply unregulated. Here, we will argue that the reply to a call needs to be regulated as well, in coordination with the regulation of the request. Of course, regulation of the reply is a *post factum* decision, in so far as the execution of the server is concerned. But such regulation can have two types of effects: (a) it can update the control state based on the nature of the reply, or on its timing; and (b) it can control the payload of the reply itself. The nature of these two types of effects, and the need for them, are discussed in the following subsections.

Updating the Control State:

We have argued that an AC policy often needs to be sensitive to the history of interaction, as represented by the control state of the policy. But under synchronous communication such interaction consists of the reply as well as the request that triggered it. The reply may be important because it may matter to the policy whether or not the server replied, how long it took it to reply, and the nature of the reply itself.

PPS policy is inherently sensitive to the reply as follows. Point 2 of this policy stipulates that payment for a service should be *moved* from the e-wallet of the client to that of the server. But this should happen only upon a successful completion of the service—that is, when the client receives a non-exception reply from the server. It is obvious that this policy can be implemented only if the reply is regulated; and if such reply control is coordinated with the control of the corresponding request.

Controlling the Payload of the Reply:

Access control policies are often concerned with what information clients are allowed to access. Often, the sensitive information disclosed to the clients becomes explicit only at the time of reply, and not at the time of the request. The reply needs to be regulated in order to control the payload itself.

To show how this control may be useful, consider an elaboration of policy PPS, via the following additional point:

> *Patient record servers may serve three kinds of clients:* doctors, *who have access to an entire patient record;* researchers, *who have access to all the information within a record, except for the patient name and id; and* financial officers, *who are not allowed to see any medical information within a record.*

This part of our policy cannot be enforced at the request time, since the patient record information is not available at that time. Only after the server replies, the complete record of the patient is available, and the appropriate fields can be filtered based on the role of the caller.

3.2.2 The Need to Regulate Timeouts

Under synchronous communication the client thread is blocked until it receives a reply. This feature is intended to provide transparency of the network communication, by making remote calls appear to programmers as local calls [13]. But this transparency is often hard to maintain in practice because the duration of a service is unpredictable, due to communication uncertainties, particularly over WANs; and due to the lack of familiarity with the behavior of the server, particularly when it belongs to a different administrative domain.

The conventional technique for dealing with such unpredictability is for the client to terminate a given service call—if it takes too long to complete—simply by killing the requesting thread. But such an arbitrary, one-sided timeout may be harmful. The problem is that both the client and the server have stakes in the service, which might be undermined by its abrupt termination, unless the termination is done in an *orderly manner*. The meaning of "orderly"

depends on the application at hand, as we shall see below. But whatever it may be, it ought to be defined explicitly in the policy regulating the communication, so that it can be enforced by the AC mechanism, and be visible to both the client and the server. There are many possible termination (timeout) policies, which may be suitable in different situations. We will consider two types of such policies below.

Predefined Timeouts:

To provide a degree of predictability to the duration of a service, one can employ a policy under which every call would specify an upper limit $Tmax$ for the duration of requested service, which would be provided to the server as a parameter. This would mean that if the reply does not arrive at the client by the specified limit, the client would regain its control, and the server will be notified of the termination (assuming that the server implements proper interfaces that support such notification). This policy benefits the server as follows: if it knows that the requested service cannot be provided within the time $Tmax$, it might decide to decline the request immediately, and not waste its resources on attempting to provide it. The client would also benefit from this policy by not having to forcefully kill the thread that issued the call—a measure that can leave the application in an inconsistent state.

Moreover, if the service in question is of a pay-per-service kind, then such a policy can mandate the return of the payment to the client, if the requested service has not been provided by the specified limit $Tmax$. This is appropriate because one can argue that the server does not deserve any payment for its effort in this case, since it has been notified *a priori* of the time limit.

The time in this policy can be strictly local, and the enforcement can be expressed in either client or server time. Distributed clocks, however, are often reasonably synchronized (using NTP, GPS, or other mechanisms), thus the two local times in practice are the same.

Unplanned Timeouts:

Sometimes, it is desirable to allow the client to interrupt a call while the call is still in progress. This may be the case if runtime conditions at the client indicate that the service it has requested is not necessary anymore, or if the thread that initiated the call needs to regain the control, for

whatever reason. But even if unplanned, such a timeout needs to be done in an orderly fashion, according to a pre-specified policy.

Point 3 of the PPS policy is a provision regulating such unplanned timeouts. This point stipulates that the server—whose work has been terminated for no fault of its own—be compensated by a specified fraction of the cost of a normal service; and that the rest of the payment be returned to the client. Thus, this policy ensures a degree of fairness to both the client and the server, whenever the client terminates its call. The implementation of this particular policy under the proposed AC mechanism is presented in Section 3.4.

3.3 Regulating Synchronous Communication

As we have already pointed out, synchronous communication differs from asynchronous one in that the former consists of two tightly coupled steps – the request and the reply – and because the client thread is blocked until it gets the reply. Conventional AC mechanisms for synchronous communication operate by regulating only its request part, usually intercepting the request at the server side, as shown in Figure 3.1. This is similar to the manner that conventional AC mechanisms for asynchronous communication operate.

In this thesis we propose a generalized regulation mechanism that controls both the request and the reply separately, but in a coordinated manner, with respect to both the client and the server. This regulation takes place in four steps, as depicted in Figure 3.2. Any request placed by a client is intercepted first by the LGI-controller associated with the client, then by the controller of the server. When the server issues the reply, it is intercepted by the controller of the server, and then by the controller of the client. Each controller enforces the same communal law $\mathcal{L}$, which can be written to coordinate the treatment of the reply with the request that triggered it via the state it maintains.

The implementation of the mechanism is discussed in Section 3.4. In this section, we will show how this mechanism can be used to implement the PPS policy. For this purpose, we will express a law that implements this policy via a pseudocode. A formal implementation is available at [64], and it uses the Java-based law language of LGI, presented in detail in chapter 5. The pseudocode follows the $\langle event, condition, action \rangle$ pattern of LGI. Controlling

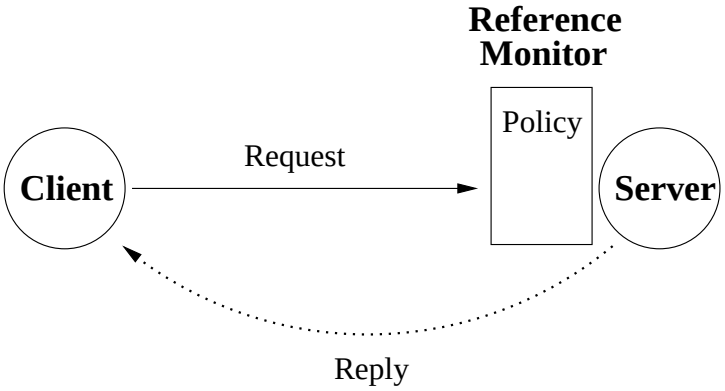

Figure 3.1: Server Centric Access Control over RPC

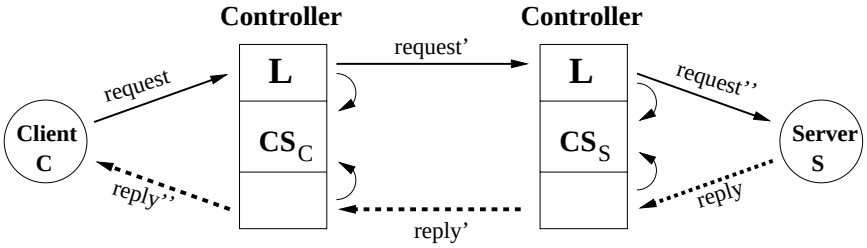

Figure 3.2: Regulated Synchronous Communication

synchronous communication, however, requires a number of different events:

- $sentCall$: occurs at the controller of the client when a client performs a request.
- $arrivedCall$: occurs at the controller of the server when a request arrives at it.
- $sentResult$: occurs at the controller of the server after the server initiates the reply.
- $arrivedResult$: occurs at the controller of the client when the reply arrives at it.

The `condition` part of a rule is an arbitrary expression defined over the identity of the caller and the callee, the payload of the request or the reply, and the local control state.

The `action` part of a rule consists of a list of operations that mandate such activities as the manipulation of the request and reply or the modification of the control state. The modification of the control state is critical in the context of synchronous communication, since it allows the

recording of the relevant aspects of the history of interaction. In particular, the state can be used to facilitate the coordination between the control of request and the control of the reply.

3.3.1 Case Studey: The Implementation of the PPS Policy

Access to services under our Pay-Per-Service (PPS) policy is regulated using a *currency* consumption scheme. The currency represents a form of credentials used for regulation purpose, thus the e-wallet of clients and servers are maintained securely by their controllers as a form of state—the control state. The budget officer is recognized as such by having its controller maintain a `role(budgetOfficer)` credential in its control state. The acquisition of this credential and the initial setup of the corresponding state can be performed using either a digital certificate, an appointment, or a password scheme; these details are not discussed further, but can be found in [45].

Figures 3.3 and 3.4 present the law implementing the PPS policy. Rules $\mathcal{R}1$ - $\mathcal{R}4$ control how the currency is distributed among the clients and servers—corresponding to Point 1 in PPS, Rule $\mathcal{R}5$ - $\mathcal{R}8$ regulates the access to the server according to the available currency—corresponding to Point 2 in PPS, and Rules $\mathcal{R}9$ - $\mathcal{R}12$ regulate the cancellation of services using an unplanned timeout mechanism corresponding to Point 3 in PPS.

Rule $\mathcal{R}1$ specifies that everybody can request a replenishment of its currency, anytime during the interaction, via a `getBudget` request. $\mathcal{R}2$ prohibits such requests to be served by anybody but a proper budget officer. This is done as follows: whenever an `arrived-Call(getBudget)` event arrives at a destination controller, the local control state is looked-up for `role(budgetOfficer)` credential. If the local state contains this credential, the target is allowed to handle the request. If not, a `NotBudgetOfficer` exception is returned to the caller. Rule $\mathcal{R}3$ allows the budget officer to reply with a certain currency amount, unhindered. Rule $\mathcal{R}4$ retrieves the assigned currency from the reply, and adds it to the e-wallet of the client. Since this currency constitutes a credential for the subsequent communication, it should be maintained by the client's controller in its state.

Rules $\mathcal{R}5$ to $\mathcal{R}8$ regulate the access of a client to a service, based on the cost of the service and the amount available in the client's e-wallet. We assume that the cost of a service is a fixed amount, denoted by the value `serviceCost`, while the name of the service (i.e remote

Preamble: Law(PPS)

$\mathcal{R}1$. `upon sentCall(getBudget)`
`do forwardCall`

Any agent is allowed to request a budget increase

$\mathcal{R}2$. `upon arrivedCall( getBudget)`
`if role == budgetOfficer`
`do forwardCall`
`else`
`do forwardResult(Exception(NotBOfficer))`

Only a `budgetOfficer` *is allowed to serve budget requests*

$\mathcal{R}3$. `upon sentResult(getBudget)`
`do forwardResult`

The budget officer can issue a reply containing a budget increase

$\mathcal{R}4$. `upon arrivedResult(getBudget)`
`do addEWallet(method.result), do forwardResult`

Upon receiving a budget increase, the `e-wallet` *stored in the control state of the agent is incremented with the amount specified in the reply.*

$\mathcal{R}5$. `upon sentCall( S )`
`if eWalletAmnt < cost`
`do forwardResult(Exception(OutOfCurrency))`
`else`
`do removeEWallet(cost),do addEscrow, do forwardCall`

Upon sending a request for service `S`*, the service cost is deducted from the e-wallet of the client, and placed under an* `escrow` *term in the control state.*

$\mathcal{R}6$. `upon arrivedCall(S)`
`do addEscrow,do forwardCall`

Upon receiving a service request, an `escrow` *containing the cost of the service is setup in the control state of the server.*

$\mathcal{R}7$. `upon sentResult( S )`
`if method.result is Exception`
`do removeEscrow,do forwardResult`
`else`
`do addEWallet(cost),do removeEscrow, do forwardResult`

If the reply is an exception, the escrow of the server is discarded; otherwise, the e-wallet of the server is increased with the amount in the escrow.

$\mathcal{R}8$. `upon arrivedResult( S )`
`if method.result is Exception`
`do addEWallet(cost),do removeEscrow, do forwardResult`
`else`
`do removeEscrow, do forwardResult`

When the service reply arrives at the client, if it represents an exception, e-wallet of the client is replenished with the amount in the escrow; otherwise the escrow is discarded.

Figure 3.3: Pay-per-service law (part I)

$\mathcal{R}$9.
```
upon sentCall(cancel)
       do forwardCall
```
An agent can cancel a pending request anytime.

$\mathcal{R}$10.
```
upon arrivedCall( cancel )
       if escrow.exists()
          do addEWallet(f(cost)), do removeEscrow
          do forwardResult, do
       forwardResult(Exception(Cancelled))
       else
          do forwardResult(Exception(NoPendingCall))
```
When the cancelation request arrives at the server, if there exists a pending request, a penalty `f(cost)` *is added in the e-wallet of the server, and replies are issued for the cancellation request and the service itself.*

$\mathcal{R}$11.
```
upon arrivedResult( cancel )
       do forwardResult
```
The reply to the cancel request is forwarded to the client

$\mathcal{R}$12.
```
upon arrivedResult( S )
       if method.result is Exception(Cancelled)
          do addEWallet(serviceCost-f(cost))
          do removeEscrow, do forwardResult
```
When the canceled service reply arrives at the client, the e-wallet of the client is replenished with the cost of the service minus the fraction penalty; the escrow is removed and the reply propagated to the client.

Figure 3.4: Pay-per-service law (part II)

method, procedure) is represented by the variable `S`. The regulation is performed in a combined manner, on the request as well as on the reply path. In rule $\mathcal{R}5$, whenever a client requests a service, the cost of the service is compared against the e-wallet of the client. If the cost exceeds the e-wallet amount, an `outOfCurrency` exception is returned to the caller. If the client has enough currency, the cost of the service is deducted from the e-wallet of the client. The state of the client is augmented with an item called *escrow*, which binds the cost with the request information (such as request_id, object_id, request signature). Finally the request is allowed to propagate. Rule $\mathcal{R}6$ occurs when the server's controller detects a service request. In this case, a similar *escrow* state is saved in the local state, on behalf of the server. Rule $\mathcal{R}7$ occurs when the server replies to the client. Remember that PPS policy specifies that only a successful service is to be paid for; the non-success is determined by the return of an exception. If such an exception occurs, then the previously setup escrow is removed without crediting the e-wallet of the server; otherwise, the e-wallet is credited with the service cost. Rule $\mathcal{R}8$ performs the corresponding activity on behalf of the client: if the result was an exception, then the client's e-wallet is credited back with the service cost and the escrow state is removed; otherwise, the service is considered successful, and the escrow state is simply removed.

Rules $\mathcal{R}9$ to $\mathcal{R}12$ correspond to the PPS cancellation of service. $\mathcal{R}9$ allows anybody to cancel a service request. Whenever such a cancellation request is sensed by the controller of the server, $\mathcal{R}10$ is fired. This rule checks whether the server has already issued a reply, by checking the escrow state. If this is the case, the cancellation request cannot be satisfied and a `NoPendingCall` exception is returned. If the server is still handling the service, then the cancellation takes effect: the escrow is removed, the e-wallet of the server is credited with a fraction of the cost (denoted by the function `f(serviceCost)`), and two replies are issued automatically, without the server's involvement. First, a successful reply to the cancellation request is issued, followed by an exceptional reply to the cancelled service (`Canceled` exception). Rule $\mathcal{R}11$ allows the cancellation reply to reach the client, while $\mathcal{R}12$ handles the situation of the `Canceled` exception reply to a service. This rule is similar to Rule $\mathcal{R}8$ that handles any reply to a service. In this situation, however, the e-wallet of the client is replenished with the cost of the service minus the fraction penalty; similarly, the escrow is removed and the reply propagated to the client.

3.4 The Implementation of Regulated RMI

In this section, we outline an implementation of the access control model for synchronous communication applied to Java Remote Method Invocation (Java RMI or simply RMI) [71]. RMI is a mechanism that allows remote procedure calls between objects located in different Java virtual machines. When a client performs a request, a method is transferred to the server along with its serialized arguments. When the server answers, the return data (or an exception) is serialized and transferred to the client. The data exchanged in this process consists of the method name and signature, along with the argument or reply objects.

The implementation presented here, called Regulated RMI (or RRMI), is a modified version of Java RMI, and is virtually source-level compatible with it. This section has three parts. The first part describes the LGI laws that regulate RMI communication (also called RMI laws). The second part describes the changes we introduced in the RMI suite. Finally the performance of RRMI is discussed.

The Formulation of RMI Laws:

In order to provide a fine-grained access to the information exchanged during an RMI method call, the RMI laws are written using Java Laws, to be discussed in details in chapter 5. The access control rules are expressed in RMI laws by mapping the events introduced in Section 3.3 to specific methods, called *event methods*. The conditions are represented by Java code operating over the local state, the method name/signature and the arguments/reply values. The actions are represented by specific methods that mandate the handling of the request/reply and the modification of the local state. Whenever an event occurs at the controller, the corresponding event method in the RMI law is invoked. The computation of such a method, in turn, produces a number of actions to be carried out by the controller. The formal RRMI law implementing the PPS policy is available at [64].

The RRMI suite:

At application level, the RRMI suite is largely compatible with Java RMI. The only difference between the two suites appears in the initialization stage, when a security principal is associated

with the stub of a caller and the skeleton of a remote object (i.e., the target of a call). The important components of the RRMI suite are as follows: RRMI has an LGI-enabled transport protocol – different from JRMP, or IIOP; there is a different stub compiler, called *LgiRMIC* instead of the standard *RMIC* compiler; a new *registry* application, *LgiRegistry* regulates the exchange of stubs between applications. Below, we discuss these components.

The JRMP transport protocol is employed in the RMI stub-to-skeleton interaction. In order to enable control over RMI communication, we devised a new transport protocol based on LGI. As opposed to JRMP, this new transport layer attaches additional information to a request and reply, enabling access control decision based on the method name, its signature, and runtime arguments.

A control decision in LGI model can be based on the identity of the interacting principals: i.e., the client and the server. In order to perform a principal-based decision, the caller and the remote object are associated with their own principals. Since the communication endpoints are the stub and the skeleton, we modified the RMI compiler in order to allow the association of a principal to each stub and skeleton. The newly resulted compiler is called *LgiRMIC*.

We also developed a new registry entity. Our LgiRegistry is an LGI-enabled repository for stubs, that offers LGI control over the propagation and publishing of remote object stubs.

Due to the nature of the above modifications, our implementation was based on NinjaRMI [50]. This is an open source RMI implementation developed as part of the Ninja project at Berkeley, and is source-level compatible with Java RMI.

Figure 3.5 presents a simple example of source code and the API provided by RRMI. In this example, *PMember* represents the principal member object, a principal subject to LGI regulation. *LgiNaming* represents the registry used to bind and lookup the published objects. The example shows the definition of a remote object, *RecordServerImpl*. The *principal* argument of the constructor establishes the identity of the principal exporting this object. The initialization and the actual exporting of the object can be observed in the server code. The client code shows the initialization of the principal performing remote calls. The actual stub for the remote object is downloaded from the *Naming* registry using the *lookup* method. This method also attaches the identity of the principal of the caller to the downloaded stub. After these steps, any remote call will carry –in a seamless manner–the identity of both the caller and the recipient of the

```
                /*remote object code*/
public class RecordServerImpl extends LgiRemoteObject implements RecordServer{
  public RecordServerImpl(PMember principal) throws RemoteException{
    super(principal);
  }
  public String getRecord() {
    ...// specific code
  }
}
                /*exporting server code*/
PMember callee = new PMember("http://lawurl","controller",port,"server").adopt();
LgiNaming Naming = new LgiNaming(callee);
RecordServer rs = new RecordServerImpl(callee);
Naming.rebind("registry_name/object_name", rs);

                  /*client code*/
PMember caller= new PMember("http://lawurl","controller",port,"client").adopt();
LgiNaming Naming = new LgiNaming(caller);
RecordServer rs = (RecordServer) Naming.lookup("registry_name/object_name");
rs.getRecord(); //remote method invocation
```

Figure 3.5: Sample RRMI client-server code

call. It can be observed that, except for the imported/used packages, the initialization of the principal, and the stub downloading, the rest of the code is source compatible with Java RMI.

Due to the similarities between the source code of an application developed with the RRMI suite and the source code of an application developed using the traditional Java RMI, the interchange between the two protocols can be performed in a straightforward manner. In order to facilitate the adoption of the RRMI suite, we have developed a tool that performs such transformations automatically. The tool, called *rrmi-transformer*, transforms a Java program that is developed with Java RMI technology into a program that uses RRMI. The tool is based on Apect-Oriented Programming [39] techniques, and uses the AspectJ weaver [9] for performing the parsing of the code. The tool can operate both on the source code as well as on the bytecode of an application.

Given a server and a client code C_s and respectively C_c that are developed using Java RMI, *rrmi-transformer* operates as follows:

- For every exported RemoteObject in C_s (i.e., the callee), the skeleton of the server object is replaced with the skeleton of the corresponding LgiRemoteObject;
- Every call to a RemoteReference in C_c (i.e., the caller) is replaced with a corresponding call to an LgiRemoteReference;

At runtime, all the LgiRemoteObjects in the address space of the server are associated with an LGI controller, thus operating as a single agent. Similarly, every LgiRemoteReference in the address space of the client is associated with a different controller, thus operating as an agent that is distinct from that of the server.

3.4.1 The Performance of RRMI

Overall Performance

We compared the performance of RRMI implementation with standard Java RMI/JRMP. The objective of our performance tests was to evaluate the overhead introduced by our mechanism compared to raw Java RMI (with *no* AC) . We measured the average completion time for RMI calls in the case of LAN and WAN networks using different scenarios. The LAN consisted of a 10Mbps Ethernet network connecting two SunUltra10 (440Mhz) workstations. For the WAN scenario, we used an additional Intel Pentium IV (1.5GHz) placed in a 100Mbs Ethernet LAN 25 hops away from the first LAN. For both scenarios we measured method calls with String and Vector arguments/return values of various sizes. In the case of Java RMI, no access control was performed, and no security manager/class loader installed. In the case of RRMI, we provided minimal control with a simple law that retrieved the method name and one argument and compared them to predefined values. In both cases, the actual implementation of the remote method was to simply return the argument.

The results in Figure 3.11 and 3.13 show the comparison between the performance of RRMI and JavaRMI/JRMP when strings of 10, 100, and 1000 characters have been sent over, and returned, as part of a method call. The graphs in Figure 3.12 and 3.14 show the same comparison when a Vector of Integers with 10, 100, and 1000 items has been sent as an argument and returned as a result.

While the LAN measurements showed our implementation to be, on average, 2 to 4 times slower than that of Java RMI/JRMP, the overhead in the case of WAN was 8% for large sets of data. In a LAN, the serialization/deserialization and extra communication are, by far, the dominant time-consuming component of an RRMI call, and our solution requires the additional marshaling and serialization operations at the controller. Additionally, Java RMI is optimized for communication of strings and small payloads, while RRMI incurs an additional penalty by carrying extra security-related payloads. As observed in Figure 3.13 and 3.14, in the case of SANs or WANs, these disadvantages are amortized by the large communication latency. Given the added value of our mechanism, the results are encouraging. At the same time, the results prove our implementation to be comparable or better than RMI/IIOP, as reported in [37], for both LAN and WAN. We also discovered that the impact of the law complexity over performance was relatively small in general (tens of μs), thus insignificant for end-to-end method calls.

A More Detailed Analysis

In order to better understand the overhead introduced by our mechanism, we have performed additional experiments. The main goal of these experiments was to break down the individual overheads introduced by the various components of RRMI and to compare them against their Java RMI counterpart, thus providing an estimate for future optimizations of RRMI, and an assessment of how our mechanism may perform in a general environment.

The experiment setup is shown in Figure 3.6: the RMI client and server run on two Linux workstation (2.8 GHz CPU, 2G RAM), placed in the same 1GBit Ethernet LAN. Additionally, in the case of RRMI we have used two controllers sharing the same controller pool running on a third workstation, with the same characteristics (2.8 GHz CPU, 2G RAM), located in the same LAN. While the co-location of the two controllers on the same machine does not reflect the general RRMI interaction model for widely distributed systems, it is consistent with a LAN deployment of our mechanism, and it helps simplify the evaluation of various overheads introduced by our mechanism.

Similar to the previous set of experiments, we have measured the time to complete a remote method invocation (also referred to as RTT). To account for various use cases, we have

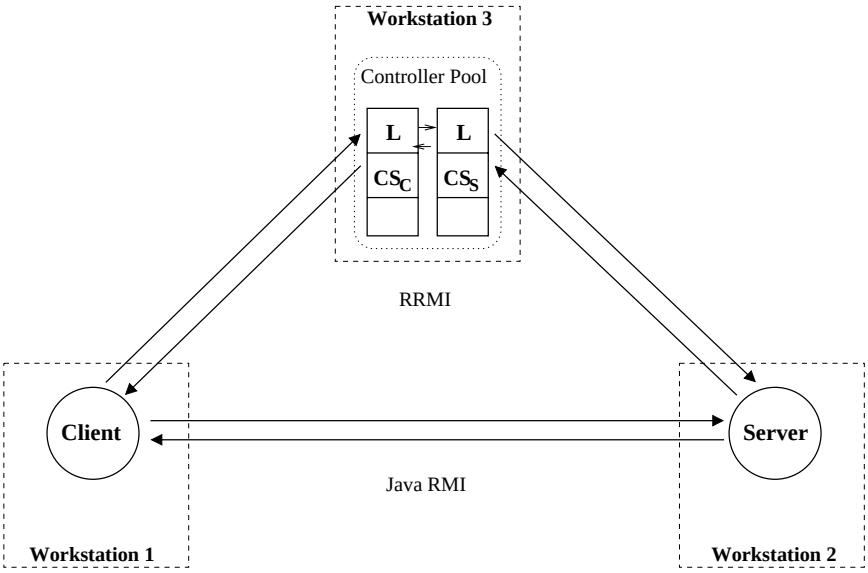

Figure 3.6: Java RMI and RRMI experiment setup

Method	Signature	Arguments	Return
m1	public void m1(void)	none	none
m2	public String m2(String)	20 character String	20 character String
m3	public Vector m3(Vector)	Vector of 500 items; Each item consists of an Integer(int) object	Vector of 500 items; Each item consists of an Integer(int) object

Figure 3.7: The characteristics of the employed method invocations

employed the method calls and arguments shown in Figure 3.7. Method `m1` has been used in order to evaluate the maximum overhead incurred for very small method calls used in signaling or synchronization between applications; method `m2` represent a more typical small request; while method `m3` reflects a more realistic use case encountered in sophisticated Java-typed data exchanges between the client and the server, with a two-level encapsulation (i.e. objects containing objects). Method `m3` reflects more closely the various policies and scenarios we have introduced in Section 3.1. Table 3.8 shows the average completion times for those methods in the case of Java RMI and RRMI.

It can be observed that in the case of small method calls, RRMI performs 8 to 10 times worse than Java RMI, while in the case of sizeable arguments/reply value, the completion time

Method	Avg. RTT Java RMI	Avg. RTT RRMI
m1	0.17618 ms	1.61115 ms
m2	0.20184 ms	1.82684 ms
m3	3.62095 ms	9.8683 ms

Figure 3.8: The average method invocation completion time

is 2 to 3 times larger than that of Java RMI.

The overhead introduced by RRMI can be explained by the following factors:

1. Extra communication paths. It can be observed that a single RMI call consists of 2 LAN communication paths: host 1 - host 2 - host 1. At the same time, an RRMI call consists of 4 LAN communication paths: host 1 - host 3 - host 2 - host 3 - host 1. This factor represents an inherent model overhead and single handedly doubles the overhead of RRMI in the case of LAN deployment.

2. Overhead in the RRMI protocol stack . RRMI sends additional data between the client and the server, via the controller. The additional data consists of information regarding the method in transit (i.e. its signature and data types), which is not transferred in the JRMP implementation of the Java RMI [71]; furthermore, LGI specific data is introduced in the exchange, such as the secure identity of the source, the destination, and law identification. Consequences of the data overhead are increasing communication time, and more importantly, a more complex and costly serialization/deserialization procedure.

3. Controller event handling and law evaluation. The controller itself introduces an overhead in managing the events and dispatching them for evaluation, as well as for carrying out the primitive operations mandate by the law.

Below we will concentrate on the first two sources of the overhead. The overhead introduced by the controller and the law will be discussed in detail in Section 5.5. Table 3.9 presents a comparison between the amounts of data generated by Java RMI and RRMI, their corresponding serialization/deserialization time, as well as the communication (RTT) time in the benchmark LAN.

Method	Data size	Avg. serialization time	Avg. communication time
m1 Java RMI	12	0.00358 ms	0.11243 ms
m1 RRMI	293	0.07011 ms	0.2075 ms
m2 Java RMI	36	0.00537 ms	0.12007 ms
m2 RRMI	391	0.09344 ms	0.2412 ms
m3 Java RMI	5378	1.1408 ms	1.2798 ms
m3 RRMI	5679	1.3029 ms	1.3359 ms

Figure 3.9: Data amount, serialization and communication time comparison

It can be observed that in the case of method m1, the serialization time increases 20 fold while the communication time increases 2 fold. Thus, for this method the overhead introduced by the controller and law evaluation is dominant accounting for $\sim$ 1 ms out of the total 1.6ms; the overhead of the communication comes second, with a total value of 2 x 0.2 ms and the overhead of the serialization in third place for a value of 4 x 0.07 ms

In the case of method m2, the increase in serialization and communication maintains the same proportions as in the case of method m1. The dominant factors are again the controller overhead, accounting for about one half of the total overhead, and the serialization and communication overhead sharing one quarter each of the overhead.

In the case of method m3, the communication becomes the dominant overhead due to the inherent proxy model, and only with a small contribution from the extra data exchanged. The serialization overhead becomes negligible, while the controller/law overhead introduces a modest overhead.

Following these results it can be inferred that the overhead for small methods can be reduced by minimizing the controller and serialization overheads. The controller overhead can be minimize when employing a powerful host for the controller, while the serialization can be improved by optimizing the RRMI implementation, in a similar manner to the optimizations used in Java RMI, such as direct stream write/read, and the avoidance of multi-layer encapsulation of arguments.

It can be observed that the controller overheads and the RRMI stack overheads measured here do not depend on a LAN or WAN deployment. Thus, a WAN deployment will have to take into account the variable communication time. Table 3.10 displays the communication

Data size	Avg. RTT 1-hop	Avg. RTT 2-hop	Avg. RTT 25-hop
12	0.11243 ms	0.27090 ms	16.778 ms
36	0.12007 ms	0.33000 ms	16.932 ms
293	0.2075 ms	0.4136 ms	17.527 ms
391	0.2412 ms	0.5267 ms	18.042 ms
5378	1.2798 ms	1.8585 ms	22.092 ms
5679	1.3359 ms	1.8992 ms	22.181 ms

Figure 3.10: Communication overhead for different networks

time for the above payloads for a 2-hop, and a 25-hop network. It can be observed that the communication time increases dramatically especially for small payloads, thus reducing the overhead of our model even when a 2-hop network is employed. When a WAN deployment of our mechanism is sought, it is recommended that the controllers are placed as closely as possible from the location of both the client and the server, thus employing a single high-latency controller-to-controller link, and two small latency client-to-controller and server-to-controller links. In such a case, the apparently irreducible two-fold overhead due to the communication time experienced in the LAN deployment can be effectively minimized. This overhead can be practically estimated by comparing the values in the first column of the table with the values in the second and third column.

Note that in our benchmarking we invoked remote methods that employ no computational overhead, in order to compare just the overhead of the protocol. When a non-trivial computation is performed at the server side, the perceived overhead of our mechanism can be significantly lower. For example, if a direct sorting of the 500 Vector values is employed in method m3, it raises the baseline of the measurements by an additional ~ 1.1 ms, thus reducing the perceiving overhead to less than 50 percent in the case of LAN, overshadowing the overhead of RRMI. Also note that the overhead introduced by the law itself is minor, accounting for less than $\sim$ 0.050 ms evaluation time, as it will be discussed later in Section 5.5.

3.5 Related Work

We are not aware of any published proposal to regulate the reply, and none of the conventional RPC-based middleware implementations provides for such regulation.

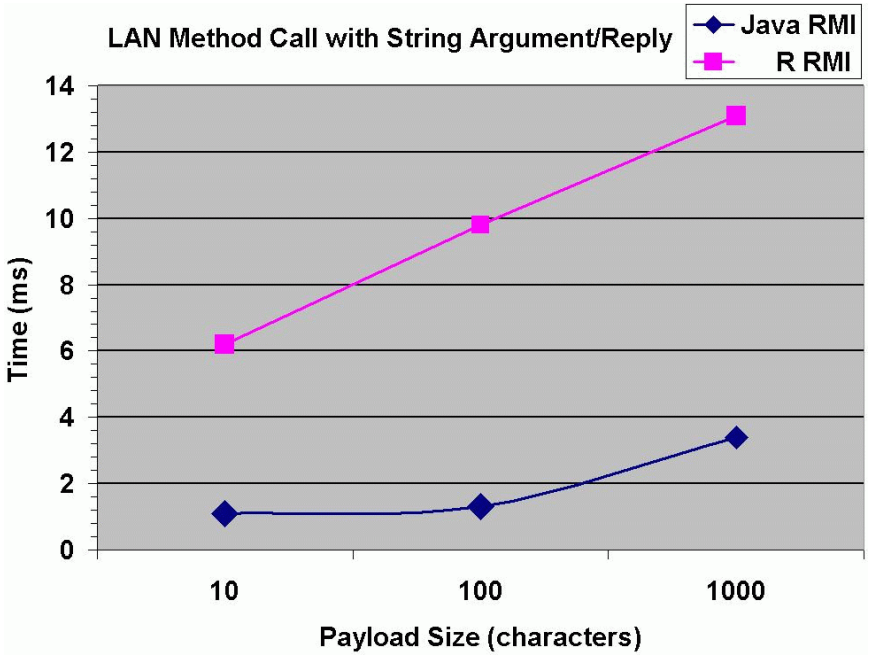

Figure 3.11: Java RMI vs. RRMI (String Transfer-LAN)

Predefined timeout is not available under Sun RPC, Java RMI [72], and DCOM [41]. These middlewares rely on the underlining network stream timeout (which is neither explicit nor predictable). Under CORBA [52, 12], a client can specify a timeout interval, but the server is not informed of it.

A number of researchers addressed the treatment of unplanned timeout, and various protocols have been proposed for that [40] [60] [28]. These protocols, however, are hard-wired in the communication mechanisms, and they provide very little flexibility with respect to the actions that can be taken by the server or the client, and the effect of these actions.

Moreover, we are not aware of any prior attempt to incorporate timeouts in any access control mechanism or in any access control decision. In our approach, the timeout and its handling are made explicit in the access control policy, thus providing the flexibility required by both the application and by the access control policy. This renders a powerful tool that can be used in client-server applications, even in the absence of communal and stateful policies.

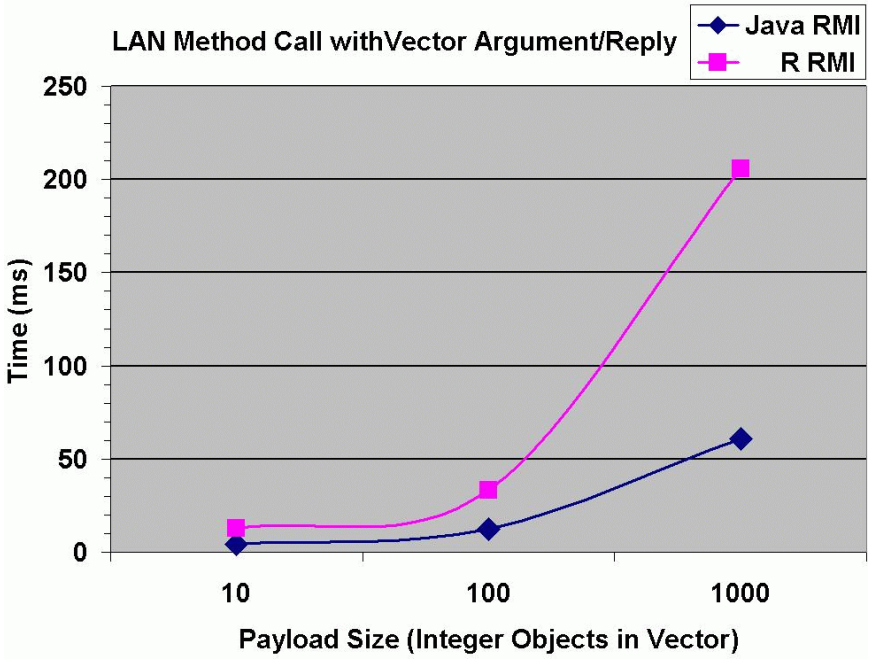

Figure 3.12: Java RMI vs. RRMI (Vector Transfer-LAN)

3.6 Summary

In this chapter, we have argued that it is necessary to employ a specialized model of access control specifically designed for synchronous communication. The primary reasons for employing such a model are the sophistication of the access control policies on the one hand, and the inherent differences between message passing and synchronous communication, which can lead to security vulnerabilities if ignored. The mechanism we have proposed exerts control both over the request and the reply, and both the server side and the client side. The control takes into account, and regulates the timing of the interaction, offering a great flexibility as well as fairness to the participants.

In order to prove our concept, we have implemented the Regulated RMI suite, a variant of Java RMI that is equipped with this advanced control mechanism. The performance measurements showed that, although our implementation produced a slowdown of the communication, the overhead was relatively small in the context of large scale applications, the target of our research. Additionally, we believe that the benefits of this protocol justify this overhead.

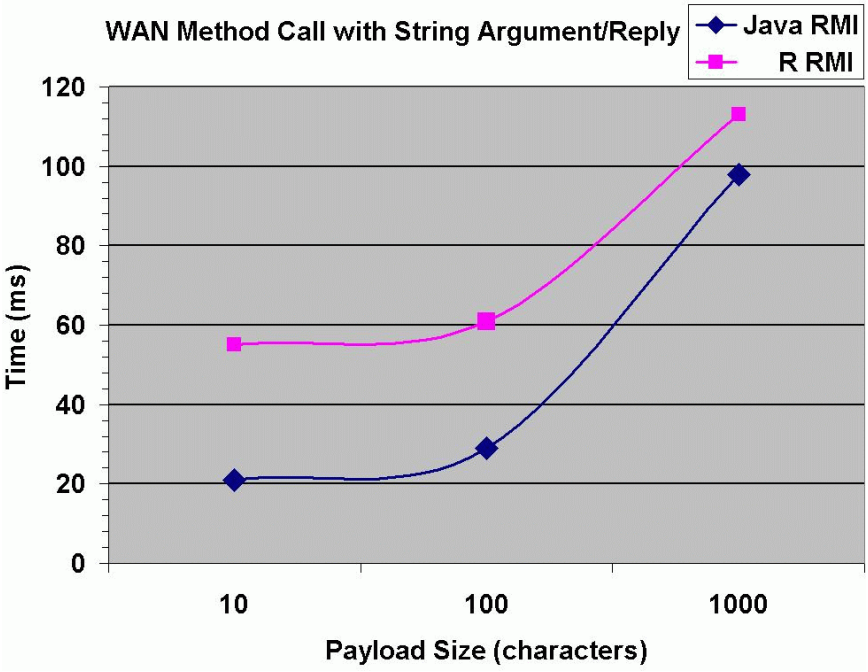

Figure 3.13: Java RMI vs. RRMI (String Transfer-WAN)

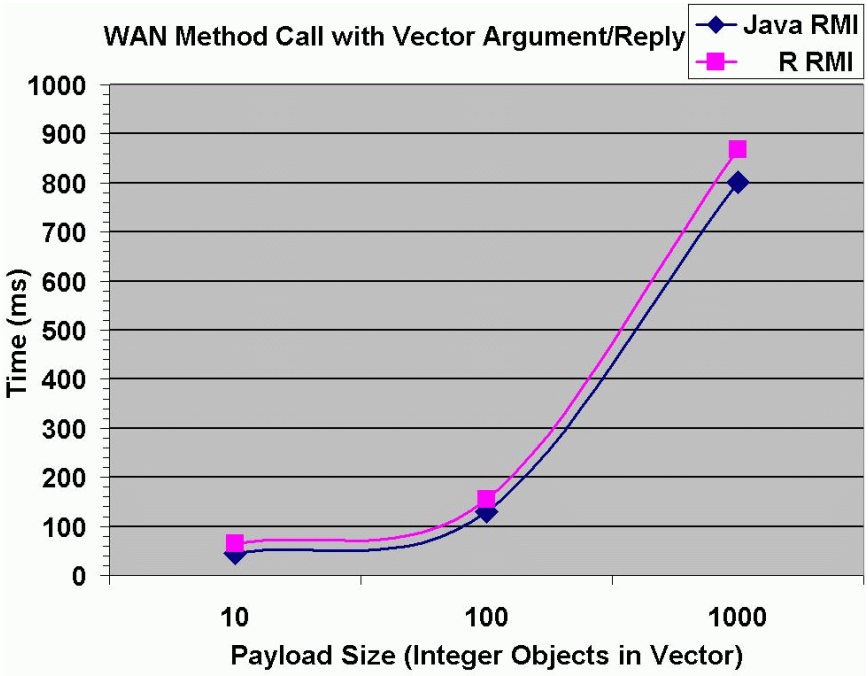

Figure 3.14: Java RMI vs. RRMI (Vector Transfer-WAN)

Chapter 4

Hot Updates

This chapter describes Hot Updates, a flexible mechanism for changing the laws of a system while the system continues to operate. We start with a motivation to provide a flexible mechanism for changing the policy in a distributed system, and we discuss the challenges for performing such an update in the case of sophisticated and distributed policies in general, and LGI laws in particular. We continue by introducing the basic mechanism that enables the Hot Updates in LGI, and describe the updating process from the point of view of an individual agent. After describing the local update, we shift our attention towards how the update takes place for an entire community in a system-wide approach. Here, we will discuss a number of methods for propagating the updates and their implication on security, performance, and applicability. We will conclude the chapter with a discussion of related work.

4.1 Motivation

The ability to seamlessly change the policy of a system while the system continues to operate is of utmost importance for high availability systems. The process of changing the policy, called the updating process, can become challenging when the policies to be substituted are computationally complex and stateful. Among the major challenges facing such an update process are the safeguarding of consistency for both the old and the new policies, as well as the need to perform a smooth, non-disruptive transition between the policies. The challenges become more obvious in the context of large and distributed systems, where the update process cannot take place atomically with respect to the whole system, due to unacceptable down-times or lack of comprehensive and centralized knowledge about all the components of the system.

In order to illustrate these challenges, let us consider an example of policy update. Consider an enterprise composed of a large number of agents that are allowed to carry internal purchasing

Preamble: Law(L_{EP0})

$\mathcal{R}$1.
```
upon sent(X,purchase(itemA), Y)
       if getBudgetA() > 0
          do decrement(budgetA), do forward
       else
          do deliver(''out of budget for item A'')
```

When an actor sends a `''purchase(itemA)''` *message, the* `budgetA` *term is retrieved from Control State. If its value is greater than 0, then it is decremented, and the purchase order is allowed. Otherwise, an* `''out of budget''` *message is delivered to the actor.*

$\mathcal{R}$2.
```
upon sent(X,purchase(itemB), Y)
       if getBudgetB() > 0
          do decrement(budgetB), do forward
       else
          do deliver(''out of budget for item B'')
```

Similar to the above rule, but for `''purchase(itemB)''` *messages. If the value of the* `budgetB` *term is 0, an* `''out of budget''` *message is delivered; otherwise, the message is forwarded.*

Figure 4.1: Law L_{EP0}

activities. In particular, the agents are allowed to order electronically two types of goods, `A` and `B`. The enterprise sets in place an initial *Enterprise Purchasing* policy (or EP_0) designed to regulate the purchasing of these items. According to this policy, each agent is allocated two separate budgets, B_A and B_B, to be used against the purchase of `A` and `B`, respectively. Whenever an agent issues a purchase order for item `A`, its budget B_A is decremented by one unit. Similarly, when the agent issues a purchase order for item `B`, its budget B_B is decremented by one unit. The budgets of each agent are initially assigned, and periodically replenished by a specialized agent—a purchasing manager.

Let us assume that, over time, EP_0 suffers a number of modifications that give rise to a new policy, EP_1. Instead of assigning separate budgets, EP_1 will define a single consolidated budget CB to be used for purchasing both items A and B. Furthermore, EP_1 will require that each agent report its budget amount periodically to the budget manager.

In order to reveal the obstacles posed by the update process, let us consider a compact implementation of these policies under LGI. Figure 4.1 shows law $\mathcal{L}_{EP0}$ implementing EP_0. The law contains two rules, each handling the purchasing of one of the items. In Rule $\mathcal{R}$1, whenever the actor sends a `''purchase(itemA)''` message, the `budgetA` term is retrieved from control state. If its value is greater than zero, then it is decremented, and the purchase order

Preamble: Law(L_{EP1})

$\mathcal{R}$1.
```
upon adopted([Args])
      do imposeObligation(reportBudget, 10)
```
Upon adoption, schedule a `reportBudget` *obligation to fire in 10 seconds.*

$\mathcal{R}$2.
```
upon sent(X,purchase(I), Y)
      if getBudgetC() > 0
         do decrement(budgetC), do forward
      else
         do deliver(``out of budget'')
```
When an actor sends a ``` ``purchase(I)'' ``` *message, where I can be either* `itemA` *or* `itemb`*, the* `budgetC` *term is retrieved from Control State. If its value is greater than 0, then it is decremented, and the purchase order is allowed; otherwise, an* ``` ``out of budget'' ``` *message is delivered to the actor.*

$\mathcal{R}$3.
```
upon obligationDue(reportBudget)
      do forward(Self,getBudgetC(), budgetManager)
      do imposeObligation(reportBudget, 10)
```
When a `reportBudget` *obligation is fired, the agent sends a message containing the current budget to the* `budgetManager`*. Subsequently, it schedules another* `reportBudget` *obligation to fire after 10 seconds.*

Figure 4.2: Law L_{EP1}

is allowed; otherwise, an ``` ``out of budget'' ``` message is delivered to the actor. Rule $\mathcal{R}2$ employs the same logic for the purchasing of item B.

The law reflecting the changes intervened in $\mathcal{L}_{EP1}$ is shown in Figure 4.2. The implementation of Rule $\mathcal{R}2$ is similar to the two rules in the previous law, except that it accommodates both items A and B, and it controls the purchasing messages based on a consolidated budget. Rule $\mathcal{R}3$ is responsible for automatically reporting the consolidated budget to the budget manager. The reporting is implemented using an obligation mechanism, a law-defined event that can be scheduled to fire at specific moments of time in the future. When a `reportBudget` obligation is fired, the agent sends a message containing the current consolidated budget to the `budgetManager` agent. Subsequently, it schedules another `reportBudget` obligation to fire after 10 seconds, effectively creating a periodically recurring event. The initial obligation is setup in the `adopted` event when the agent is created, as shown in Rule $\mathcal{R}1$.

In the interest of brevity, both $\mathcal{L}_{EP0}$ and $\mathcal{L}_{EP1}$ presented above do not show the rules for controlling the assignment of budgets B_A, B_B, and CB respectively. Furthermore, we assume that $\mathcal{L}_{EP0}$ and $\mathcal{L}_{EP1}$ contain additional rules to allow individual agents to communicate with

each other; such rules are not shown here.

In a simplistic approach, in order to perform the transition from $\mathcal{L}_{EP0}$ to $\mathcal{L}_{EP1}$, one might directly substitute the laws for each agent in the enterprise. This would, however, be a far cry from the desirable seamless transition. A minimal provision would be to preserve the budgets B_A and B_B assigned to each agent under $\mathcal{L}_{EP0}$, and transfer them under the consolidated budget CB mandated by $\mathcal{L}_{EP1}$. Otherwise, the update of the law would automatically disable the entire purchasing activity until new consolidate budgets can be reassigned by the budget officer. Thus, one general requirement of an update process is to provide a transformation between the states of the agent before, and after the transition.

While a transformation function might be necessary to provide the transfer of state during the update process, it is certainly not sufficient for a smooth transition. Let us assume that the update occurs while $\mathcal{L}_{EP0}$ is in the middle of evaluating Rule $\mathcal{R}1$, after `budgetA` has been decremented, but before the purchase order has been forwarded. Accordingly, the value of the consolidated budget after the update will reflect the decreased `budgetA` without an actual corresponding purchase order. This situation reflects an inconsistency in the control state determined by an unbalanced budget. Of course, this situation can be avoided by refusing the updates to take place while a ruling is in progress. But, this hardly solves the problem of inconsistencies, since a control state can become consistent across multiple evaluations. This is the case, for example, when a server receives a conditional payment contingent upon a successful service. If the update takes place after the evaluation of the request, when the control state reflects the payment, but before the arrival of an un-successful answer, the update would propagate an illegitimate amount into the new state, thus producing an inconsistency. In order to prevent the arise of various inconsistencies, the update should take place at well defined moments during the course of policy evaluation. These moments, as well as the semantics of consistency for a given policy, however, are policy dependent; thus, the update process should be flexible and policy dependent itself.

Beside the challenges posed by maintaining consistency at the state level for individual agents, additional challenges arise when considering the update of an entire community. As it is not feasible to carry out the update process atomically for an entire community, inevitably,

there will be agents operating under different versions of the policy at the same time. The communication of such agents can be a source of inconsistency in itself, as different policies can be aware of, and are expected to handle, only a well-defined set of messages. The closures of such message sets might not necessary be a perfect overlap between the updating policies. Thus, communication among agents and the propagation of an update in a system are inextricably connected. Additional challenges include such aspects as: who can initiate a transition; and what is a secure way of unfolding an update process throughout the community.

In the rest of this chapter, we will present Hot Updates, the process of changing laws in LGI, a mechanism that attempts to answer these questions.

4.2 The Basic Model of the Hot Updates

4.2.1 Notation and Terminology

Before describing the hot update process, let us introduce the following notation:

- A *lineage of laws* $\mathcal{L}_0$, $\mathcal{L}_1$,...$\mathcal{L}_i$, $\mathcal{L}_{i+1}$...$\mathcal{L}_n$ represents a sequence of laws a community is operating under at various stages of life; the transition from $\mathcal{L}_i$ to $\mathcal{L}_{i+1}$, $\vee\ i$ in $1..n$, represents a hot update in this lineage. In the previous example, the lineage consisted of two laws, $\mathcal{L}_{EP0}$ and $\mathcal{L}_{EP1}$.

- A law $\mathcal{L}_i$ in a lineage is referred to as the version i of law $\mathcal{L}$. An agent X operating under a law $\mathcal{L}_i$ is denoted as $X/\mathcal{L}_i$.

- Given a law $\mathcal{L}$ in the lineage, $\mathcal{L}^-$ denotes the previous law in the lineage, and $\mathcal{L}^+$ denotes the next law in the lineage. $\mathcal{L}^{--}$ denotes *any* prior law in the lineage, while $\mathcal{L}^{++}$ denotes *any* subsequent law in the lineage.

- Each law in a lineage is available, and can be accessed, at a given URL. The URL where $\mathcal{L}_i$ resides is denoted as U_i.

- A law $\mathcal{L}_i$ defines itself to be part of a certain lineage by declaring the term `previous` `(`U_{i-1}`)` in the preamble, where U_{i-1} represents the URL of $\mathcal{L}_i^-$. This term, which is present in all the laws in a lineage (except $\mathcal{L}_0$) recursively defines the lineage.

$\mathcal{R}3$. upon arrived(X, pleaseUpdate(URL), Y)
do **update(URL)**

Upon receiving a ''pleaseUpdate'' message containing the URL of the new law, the agent proceeds to update immediately.

Figure 4.3: Update Support in Law L_{EP0}

Preamble:
Law(L_{EP1})
previous(url(L_{EP0}))

$\mathcal{R}4$. upon lawChanged(L_0, L_1)
do add(budgetC(getBudgetA()+getBudgetB()))
do remove(budgetA), do remove(budgetB)
do imposeObligation(reportBudget, 10)

Upon a ''lawChanged'' *event the* budgetA *and* budgetB *terms are consolidated into a* budgetC *term, and a* reportBudget *obligation is imposed.*

Figure 4.4: The lawChanged Event in Law L_{EP1}

- Beside its own hash, a law $\mathcal{L}_i$ also maintains the sequence of hashes for all the previous laws in the lineage [$H(\mathcal{L}_{i-1})$,$H(\mathcal{L}_{i-2})$,...,$H(\mathcal{L}_0)$]. Accordingly, a law is able to identify that a given hash represents a certain law in its own lineage.

4.2.2 The Update Mechanism

One of the goals of Hot Updates is to have a smooth transition from an old law L to a newer law L^+, with as little disruption as possible for both the individual agent and for the community as a whole. Disruptions can occur due to various reasons. First, if the update occurs when an agent is engaged in certain transactions, then unforeseen inconsistencies might affect both the agent and the transaction partners. Consequently, the moment a hot update takes place at an agent impacts the course of the update process. Second, law L^+ is often a logical continuation of L; as such, L^+ has to operate in the context created by L. In order to address the above concerns, the hot updates of laws are performed with law support. First, law L retains the control over how and when it can be updated. Second, L^+ is offered the means to resolve possible inconsistencies brought about by the update process, and cope with legacy aspects of L. The rest of this section presents the detailed steps, from the initiation of an update at an individual agent, until the update is completed and the inconsistencies resolved.

The Update Primitive Operation

The hot update of a law L is triggered when L invokes the primitive operation `update(URL)`. This operation takes a single argument — the URL of the new law L^+. The `update` operation can be invoked in the context of any regulated event, effectively granting L full control to choose the best moment and context to carry out the transition to L^+.

Once the `update` primitive operation has been invoked, the updating process starts to take place. The first step is to load the new law L^+ from the provided URL. When loading the new law, the controller verifies that L^+ belongs to L's lineage by checking the `previous(`U`)` declaration in L^+.

Figure 4.3 shows how the `update` primitive operation triggers the transition between $\mathcal{L}_{EP0}$ and $\mathcal{L}_{EP1}$. The example assumes that the agent receives a message of type `pleaseUpdate(URL)`, containing the URL of $\mathcal{L}_{EP1}$. Upon arrival, the hot update is triggered immediately. The issue of how an agent receives the `pleaseUpdate(URL)` message, represents a different aspect of the update process to be covered later in Section 4.3. In general, the update primitive operation does not have to be carried out as soon as L finds out about the new law; L can choose, at its discretion, when the update is to be carried out.

The *LawChanged* Event

After L^+ has replaced L, a `lawChanged(`L`,` L^+`)` event will be invoked automatically, as the first event to take place after the update, in the context of the new law. The arguments of this event represent the old law that has been updated, and the new, i.e., current, law. The purpose of this event is to provide a mechanism to:

- i) adapt the control state of the agent X as generated under L (CS_x^L) to a new control state more suitable to L^+ (CS_x^{L+});
- ii) initialize certain required actions, effectively marking the beginning of X operating under L^+.

In general, the mapping of the control state takes place through a series of control state-related primitive operations. This event can also invoke primitive operations that change the

DCS (distinguished control state) in order to affect pending obligations, or existing authorities or portals. When a hot update takes place, the `adopted` event that usually marks the beginning of operations under a law is not triggered anymore; the `lawChanged` event can be viewed as a correspondent and replacement of the `adopted` event.

Figure 4.4 shows how the `lawChanged` event is used to adapt the control state and initialize the operations of $\mathcal{L}_{EP1}$, following a transition from $\mathcal{L}_{EP0}$. In Rule $\mathcal{R}4$, the law retrieves the terms `budgetA` and `budgetB` generated under $\mathcal{L}_{EP0}$, and creates a new `budgetC` term, representing the sum of the previous terms. Subsequently, the rule deletes `budgetA` and `budgetB` from the control state, since they are no longer used in $\mathcal{L}_{EP1}$. Also, the rule imposes a `reportBudget` obligation in order to trigger the periodic reporting of the budget to the budget manager, as implemented in Rule $\mathcal{R}3$ in Figure 4.2. Figure 4.4 shows how $\mathcal{L}_{EP1}$ declares its lineage using the `previous()` preamble declaration.

The general assumption for the Hot Updates mechanism is that L^+ represents a logical continuation of L, thus the Control State maintained under L is considered relevant to L^+. Accordingly, the `update` primitive operation propagates the Control State unchanged, and any required adaptation can be performed in the context of the `lawChanged`. If the L^+ and L are not logically related, or L's Control State is not relevant to L^+ then the `lawChanged` event can reinitialize the Control State using the `replaceCS` primitive operation.

4.2.3 Life After the Update

The hot update of a law is not an atomic action with respect to an entire community. Accordingly, after an agent has completed its transition from law L to L^+, it is possible to engage in interactions with agents that have not yet updated the law. Such interactions may become a source of inconsistencies, thus they demand special handling.

Confronting Ghosts from the Past

`Ghosts` represent events that were intended to be handled under the law L, but, due to a hot update, they are scheduled and handled under law L^+. Ghost events can appear under the following circumstances:

- `Regular messages submitted under an old law.` Consider two agents X and Y initially operating under law L (X/L and Y/L). Assume that X subsequently updates to L^+; Y, however, does not update, and it is not aware of X's transition. Furthermore, assume that Y/L attempts to send a message to X/L. When the message arrives at X/L^+, it triggers a ghost event. The message will be referred to as a ghost message.

- `Exceptions and obligations initiated before an update.` Ghost exceptions can occur as follows: assume an agent X/L submits a message to some other agent, but the message fails, subsequently causing an exception; if X updates to L^+ while either the message or the exception are in transit, the arrival of the exception will trigger a ghost exception event. Ghost obligations can occur as follows: assume an agent X/L sets up (`imposes`) an obligation to fire at a later moment of time $t + \Delta t$; if$X$ updates to L^+ before the Δt period has elapsed, the firing of the obligation at $t + \Delta t$ will cause a ghost obligation.

- `Events scheduled but not yet evaluated`. If a number of events occur at an agent simultaneously, the events are executed sequentially using a certain scheduling priority. If the law is updated while such events are pending evaluation, the *lawChanged* event takes precedence over all the other pending events; all the pending events, of any type, will be executed as ghost events after the update has taken place.

The syntax of the ghost event is `ghost(Event,` L^{--}`)`, where `Event` represents the underlying event causing the ghost and L^{--} represents the previous law where the event has been initiated. Possible actions that can follow a ghost event are: compensations that can maintain invariants of the old and/or new law; and notifications to the transaction partners that a law update has taken place. An example of notification following a ghost message will be discussed in Section 4.3. Below, we will present a compensative action.

Consider that $\mathcal{L}_{EP0}$ employs the following mechanism to assign budgets to different agents engaged in purchasing activities: a budget manager sends `assign(BudgetType,Amount)` messages to each agent, where `BudgetType` is either `budgetA` or `budgetB`, and `Amount` represents the assigned amount. Furthermore, assume that an agent updates its law to $\mathcal{L}_{EP1}$,

Preamble: Law(L_{EP1})

$\mathcal{R}5$. upon ghost(Event, L_{EP0})
if
Event==arrived(budgetManager,assign(BudgetType,Amnt),Y)
do inc(budgetC, Amnt)

if a ''ghost'' event appears due to a legacy message of the form assign(BudgetType, Amnt) *that has been initiated under L_{EP0}, then the consolidated budget is incremented with the assigned amount.*

Figure 4.5: Handling of Ghost Events in Law L_{EP1}

while such a legacy message is in flight. In order to avoid a loss of balance and deterministically maintain the total budget amount throughout the community, $\mathcal{L}_{EP1}$ has to consider such ghost event and account for the budget assigned under the previous law. Figure 4.5 displays Rule $\mathcal{R}5$ handling this situation. If a ''ghost'' event appears, and the underlying cause is a message of the form assign(BudgetType, Amount) that has been initiated under L_{EP0}, then the amount assigned in the budget for a particular item is retrieved and the consolidated budget is incremented with the corresponding amount.

Sending Messages to Agents Not Yet Updated

After updating the law from L to L^+, an agent has to face a period of uncertainty with respect to the law other agents are operating under. This situation is due to the fact that different agents may update their law at a different moments of time. Thus, an agent X/L^+ sending a message to agent Y does not know whether Y operates under L or it has already updated to L^+. If X falsely assumes that Y has transitioned, but Y still operates under L, this will cause an invalid message. Such a message, which we call a *precognition* message, is the opposite of a ghost message: it travels from a future law towards an older law. The message is considered invalid, since it can break the contract of both L and L^+, thus it can not be delivered to its destination.

In order to cope with this situation, the sender of the message is notified with a specific exception. Thus, when X/L^+ sends an M message to Y/L, where M is intended to an L^+ destination, then the message fails and an exception(X, M, Y, destinationLawObsolete(L)) event is triggered at the source of the message. Consequently, X and L^+ are given the opportunity to cope with this situation. Possible actions are: re-sending the message after

a certain period of time, under the expectation that Y will be updated; or informing Y that an update is available in order to speed up its update. The last measure will be discussed in more detail in Section 4.3.

4.2.4 Skipping a Number of Generations

If the updates in a certain lineage are issued with high frequency it might be possible that an agent X will become aware of an L^{++} update before updating to L^{+}. The hot update mechanism allows L to initiate a transition directly to L^{++}, thus skipping a number of generations. In order to provide a consistent mapping of Control State from L to L^{++} and a proper initialization, the intermediary transitions will occur implicitly. When X/L invokes `doUpdate(`L^{++}`)`, the following steps take place:

- i) the controller verifies that L^{++} represents a subsequent version of L;
- ii) the controller loads all the intermediary laws up to, and including, the final target law [L^{+}, ...L^{++}];
- iii) a sequence of `lawChanged` events is scheduled and evaluated for all the transitions in the law sequence;
- iiii) any event that might occur during the evaluation of the `lawChanged` sequence will become a ghost event scheduled under law L^{++}.

When updates are issued with high frequency, it is also possible to have ghost and precognition messages between non-sequential laws in a lineage. In order to cope with these situations, both the `ghost` event and the `destinationLaw-Obsolete` exception provide a parameter that identifies the actual law that has caused the event. Accordingly, more appropriate actions can be taken in order to recover from a possible inconsistency.

4.3 A Community's Perspective

In the previous section, we have shown how a law can trigger an update, and how this update takes place for an individual agent; so far, we have ignored how the updating process propagates throughout a community. From the point of view of the community, a law update can be viewed as a sequence of individual updates taking place at every agent in the community. The order of these updates, the initiation of this process, as well as the resolution of possible inconsistencies that might appear while the process unfolds, are all application-dependent. Accordingly, the dissemination of the new law to the individual agents is a process that has to be managed with law support. Below, we will present a number of methods to perform a law update for an entire community.

Off-Line Propagation

In this model, the responsibility to discover that a new update is available lies entirely with the actor. The actor can use any off-line mechanism to find out the availability and the URL of the new law. Subsequently, the actor can submit an update request to its controller whenever it sees fit. Figure 4.6 shows the rule of $\mathcal{L}_{EP0}$ that enables such update.

While this approach has the advantage of simplicity, it puts an undue burden on the actor. Furthermore, the update process might become lengthy, with update times varying widely throughout the community, and likely yielding inconsistencies, as discussed in the previous section.

Centralized Push

In order to have a faster update process, with all the agents updating at approximately the same time, a centralized approach might be used. In this case, a single specialized agent, or update manager, is responsible for tracking the new versions of the law and for informing all the agents in the community once an update becomes available. In order to reach all the agents in the community, the update manager has to maintain the dynamic membership of the community.

Figure 4.7 shows the alternative rules of $\mathcal{L}_{EP0}$ that enable a centralized push update. Rule $\mathcal{R}3$ and $\mathcal{R}4$ are responsible for maintaining the membership. In Rule $\mathcal{R}3$, whenever an

$\mathcal{R}3$. upon sent(X, pleaseUpdate(URL), X)
do **update(URL)**

When the actor decides to perform an update, it will send a ``pleaseUpdate(URL)'' *message to itself, where* URL *represents the address of the new law. The rule will trigger the* update *primitive operation immediately.*

Figure 4.6: Actor-directed update of L_{EP0}

$\mathcal{R}3$. upon adopted([Args])
do forward(Self, 'register', 'Update Manager')

$\mathcal{R}4$. upon quit()
do forward(Self, 'unregister', 'Update Manager')

When an agent is created, an automatic 'register' message is forwarded to an 'Update Manager'. When the agent quits the community, a corresponding 'unregister' message is issued.

$\mathcal{R}5$. upon arrived('Update Manager', pleaseUpdate(URL), X)
do **update(URL)**

When the agent receives a ``pleaseUpdate(URL)'' *message from the 'Update Manager', it proceeds to update immediately.*

Figure 4.7: Centralized push update of L_{EP0}

$\mathcal{R}3$. upon obligationDue(checkUpdate)
do imposeObligation(checkUpdate, 60)
do forward(Self, 'getUpdate', 'Update Manager')

A periodic checkUpdate *obligation is fired periodically every 60 seconds, and the agent sends a* getUpdate *message automatically to the* Update Manager *in order to inquire for an update*

$\mathcal{R}4$. upon arrived('Update Manager', pleaseUpdate(URL), X)
do **update(URL)**

When the agent receives a ``pleaseUpdate(URL)'' *message from the 'Update Manager', it proceeds to update immediately.*

Figure 4.8: Centralized pull update of L_{EP0}

agent becomes a member of the community, a `register` message is issued for the `update manager`. In Rule $\mathcal{R}4$, whenever the agent quits the community, an `unregister` message is issued. Rule $\mathcal{R}5$ is responsible for carrying out the update. This rule is invoked whenever the agent receives a `pleaseUpdate(URL)` message from the `update manager`. This law assumes the `update manager` to maintain the community membership properly, based on the `register` and `unregister` messages; it also assumes that the `update manager` sends `pleaseUpdate(URL)` messages to all the agents as soon as an update becomes available.

The centralized push approach offers the advantage of a fast, flash update with respect to an entire community. However, maintaining the membership of the community is not always possible, nor desirable, as it may conflict with anonymity guarantees specific to certain communities. Moreover, a centralized membership server might become a bottleneck for a sufficiently large community where membership changes frequently.

Centralized Pull

In order to avoid the shortcomings of the centralized push update, a centralized pull might be employed instead. This approach relies on a similar update manager that is responsible for tracking the new versions of the law. Unlike the centralized push solution, however, the update manager is not responsible for informing the agents when an update becomes available. This responsibility is shifted towards the individual agents, which are required to contact the update manager regularly in order to inquire about new updates. As a consequence, the update manager is not required to maintain the membership of the community anymore.

Figure 4.8 shows the rules of $\mathcal{L}_{EP0}$ for a centralized pull update. Rule $\mathcal{R}3$ exhibits a `checkUpdate` obligation that fires periodically every 60 seconds. As a result of this obligation, an automatic `getUpdate` request is directed towards the `Update Manager`. If the `Update Manager` decides that a new update is available for that particular agent, it will replay with a `pleaseUpdate(URL)` message. Rule $\mathcal{R}4$ shows how the update takes place when this message arrives at the agent. For brevity, Figure 4.8 does not show how the initial obligation is setup.

While the centralized pull update does not pose such restrictions on the anonymity of the

$\mathcal{R}3$. upon arrived([X,L], pleaseUpdate(URL), Y)
do add(newLaw(URL))
do **update(URL)**

When the agent receives a ``pleaseUpdate(URL)'' *message from some other agent operating on a different law, it saves the URL of the new law in the Control State and proceeds to update.*

Figure 4.9: P2P update of L_{EP0}

$\mathcal{R}6$. upon ghost(Event, L_{EP0})
if Event==arrived(X,M,Y)
URL = CS.get(newLaw(-))
do forward(Y, pleaseUpdate(URL), [X,L_{EP0}])
....

if a ``ghost'' *event appears due to a message initiated under* L_{EP0}*, then retrieve the URL of the current law from the Control State and send a* pleaseUpdate(URL) *message to the source. Apply other compensative measures related to the ghost message.*

$\mathcal{R}7$. upon exception(X, M, Y, destinationLawObsolete(L_{EP0}))
URL = CS.get(newLaw(-))
do forward(Y, pleaseUpdate(URL),[X,L_{EP0}])
...

When an exception is raised due to a prior message sent to a not-yet-updated destination, retrieve the URL of the current law from the Control State and send a pleaseUpdate(URL) *message to the peer. Additionally apply other measures related to the exception.*

Figure 4.10: P2P update support in L_{EP1}

users and does not require prior registration, it has a number of disadvantages. First, this solution generates additional traffic that can place a high burden on the update manager. Even worse, this traffic is often unnecessary, especially when hot updates are not issued frequently. Reducing the frequency of the `getUpdate` messages, is however, not a solution: a less frequent `checkUpdate` obligation can lead to a slow update process with respect to the community, once a new update becomes available.

Peer-to-Peer Update

The two previous update methods introduce an unnecessary level of centralization in a community, at odds with the decentralized and anonymous nature of interaction in LGI. Instead of having a single entity inform all other agents about an update, the peer-to-peer (P2P) update method relies on individual agents to inform each other that a new update is available. In this approach, it is sufficient to explicitly inform a single agent that a new update is available. This agent can discover, during routine communication, that other agents are not updated, and consequently it will inform them about the new law. In turn, these agents can inform their peers about the new law, effectively propagating the update throughout a community in a gossip-like manner. The P2P update approach uses the ghost and exception mechanisms introduced in Section 4.2 to discover that a peer agent is not yet updated. Unlike the previous update approaches, the P2P update requires the new law to be aware of, and provides support for, its own propagation. Thus, the new law is responsible for both detecting that other agents are operating under the old law, and for propagating the new law to the agents not yet updated.

Figure 4.9 and 4.10 show how the support of the hot update is split between $\mathcal{L}_{EP0}$ and $\mathcal{L}_{EP1}$. In Figure 4.9, an agent accepts `pleaseUpdate(URL)` messages from agents that have already updated their law. When such a message arrives, the agent saves the URL of the new law in the Control State and proceeds to update. The URL of the new law will become necessary later under the new law, in order to inform other agents of the required update. The arrived event specifies that the sender of the `pleaseUpdate(URL)` message operates under a *generic* law L, instead of the specific law $\mathcal{L}_{EP1}$. The reason for this is the fact that at the time of writing the law $\mathcal{L}_{EP0}$, law $\mathcal{L}_{EP1}$ is presumably not yet drafted, thus unidentifiable.

Figure 4.10 shows the rules of $\mathcal{L}_{EP1}$ that support the propagation of the update. Rules $\mathcal{R}6$

and $\mathcal{R}7$ employ the ghost and exception events as the basic mechanism for detecting that a peer needs an update. Rule $\mathcal{R}6$ is invoked whenever an agent operating under $\mathcal{L}_{EP1}$ receives a message from a not-yet-updated agent. Rule $\mathcal{R}7$ is invoked whenever an agent operating under $\mathcal{L}_{EP1}$ sends a message to a not-yet-updated agent. Both rules first retrieve the previously saved URL from the control state, then a `pleaseUpdate(URL)` message is forwarded to the peer; this message is to be received in Rule $\mathcal{R}3$ in Figure 4.9. Beside propagating the update, these rules are also responsible for applying other law-specific measures related to the ghost and exception messages.

In general, the P2P method of propagating the updates based on the runtime communication is not guaranteed to inform all the agents in a community about the update. If an agent, or group of agents is separated from the rest of the community, the update will not be able to propagate beyond this so-called "quarantine" line. However, it can be argued that in general, the effects of inconsistencies due to a partially updated community are minimal when there is no communication between the different parts of the system.

4.3.1 Trusting the New Law

In the case of the centralized approaches, the update of the law is at the discretion of the update manager. It is this manager that chooses the law and propagates it to all the agents in the community. Accordingly, the agents are required to trust the manager to perform these actions properly, essentially regarding the manager as trusted computing base.

In the P2P case, however, an actor has to trust the other actors to submit a proper law. A malicious actor, however, can introduce a new law in the community, and subsequently convince other agents to adopt it. At its extreme, an agent can hijack the entire community by replacing its law with any law that can bring him future benefits.

The root cause of this problem resides in the fact that a law initiates an update based on a `pleaseUpdate(URL)` message it receives from any agent operating under *any* law, as shown in Figure 4.9. Accordingly, the receiver has no basis to trust how this message is issued or whether the URL holds a legitimate update of the current law.

In order to avoid this risk, we employ the following mechanism. A law will update itself only if the new version of the law is undersigned by a trusted certifying authority. Accordingly,

$\mathcal{P}$*reamble:*
Law(L_{EP0})
authority(EP-CA, keyHash(-))

$\mathcal{R}$3. `upon arrived(X, CertifiedStatement, [Y,L])`
`if M.CA == EP-CA && M.statement == newLaw(URL)`
`do add(newLaw(CertifiedStatement))`
`do` **update(URL)**

If the agent receives a `CertifiedStatement` *message; and if the message is signed by the* `EP-CA` *certifying authority; and if the message statement contains the address of the new law, then proceed to update the law. Also save the certified statement containing the URL of the new law in the Control State.*

Figure 4.11: Trusted P2P update of L_{EP0}

$\mathcal{R}$6.

$\mathcal{R}$7.
`CertifiedStatement = CS.get(newLaw(-))`
`do forward(Y, CertifiedStatement, [X,`L_{EP0}`])`

When detecting that a peer is not-yet-updated, instead of sending a `pleaseUpdate` *message, send the* `CertifiedStatement` *previously saved in the Control State.*

Figure 4.12: Trusted P2P update support in L_{EP1}

it will not have to trust the sender of the message, or its law, but it will have to verify that the new law is signed by the trusted certifying authority. This strategy is depicted in Figure 4.11, and it is designed to replace Rule $\mathcal{R}$3 in Figure 4.9. The preamble of the law identifies `EP-CA` as a trusted authority that signs newer versions of the Enterprise Purchasing law. The CA is recognized by the hash of its public key. Upon receiving a `CertifiedStatement` object, the controller will verify the signature against the specified public key. In Rule $\mathcal{R}$3, the law checks that the public key of the signer belongs to the declared `EP-CA` authority, and tests that the assertion is `newLaw(URL)`. If positive, the enclosed URL is trusted to represent a legitimate newer version of this law. Accordingly, similar to the untrusted P2P implementation, the rule will save the `CertifiedStatement` object in the control state and will proceed to update the law. In order to enable the use of CertifiedStatements, Rules $\mathcal{R}$6 and $\mathcal{R}$7 in Figure 4.10 have to be rewritten as depicted in Figure 4.12. Instead of simply sending the URL of the new law when detecting an agent operating under the obsolete law, these rules will send the `CertifiedStatement` object, previously saved in the control state, as described above.

4.4 Limitations of the Updating Mechanism

The previous section showed a number of different methods for propagating the law update throughout a community. Although at the agent level these methods use the same basic tools to carry out the update, the effects of the update process can be significantly different, especially when considering ghost and precognition messages. It can be observed that certain methods, such as the off-line propagation and the centralized pull, are predisposed to yield such messages, while the centralized push and the P2P distribution limit them either in time, or in number. Ghost events can be seen as baggage carried from a previous life. For certain laws, the handling of ghost events might become overly burdening, effectively reducing the clarity and the coherence of the new law. Accordingly, minimizing the possibility of ghost events might become a criteria for choosing a particular approach for updating the law.

Carefully choosing the moment to perform an update with respect to both the agent and the community can also help reduce the possible ghost events that propagate to the new law. If the communication between agents is conducted using deterministic patterns, then choosing a "quiet environment" for performing the update for the community eliminates the possibility of communication between agents with different versions of the law. But this is not always possible. In certain applications, and for certain laws, it might be very difficult to find such a "quiet" moment, thus the ghost events might not be avoided altogether. This represents a tradeoff when designing both the old law and the new law, and it requires a certain degree of anticipation with respect to how the law will more likely be modified. The study of this tradeoff, as well as the possible improvements at the mechanism level that reduce the prevalence of ghost events, is a matter of future work.

4.5 Related Work

Despite the importance of access control and high-availability requirements for distributed systems, the problem of updating the policy while the system continues to operate has not been addressed extensively so far. The importance of on-line updating the law of a system had been initially recognized in [43], where a particular mechanism for such updates had been proposed. The update mechanism is divided into two stages. In the first stage, similar to the two-phase

commit protocol, the agents agree to perform an update. The second stage is divided into three parts: a relaxation period, designed to allow agents to finish activities under the old law; a passive period, designed to eliminate ghost messages; and finally the update itself, where the new law is set in place along with a new control state. There are many differences between the mechanism proposed in [43] and our mechanism. Most important, in our work it is the law that decides—in a flexible manner—under which circumstances to carry out an update, and not an agent at the proposition of a centralized entity. Second, our mechanism allows various models of propagating an update throughout a community, of which four particular cases had been presented. Our mechanism is not bound to a particular propagation model, nor to a specialized agent to initiate and direct the update; the importance of maintaining this flexibility had been emphasized in Section 4.1.

The Hot Updates model presented in our work has a number of affinities with the concept of automatically upgrading a distributed systems. Most prominently, the model of Ajmani et al. [1, 2] shares a number of features, such as the peer-to-peer automatic discovery of inconsistencies between different versions of a software; a delayed, and controlled scheduling of an update, as well as the mapping of the state as part of the update process. There is a number of significant differences, however, between their model and our model. First their model is valid only under the assumption that a distributed node consists of a single, updateable, object. An extension allowing multiple co-located objects would promote inconsistencies created by uncontrollable interaction between updated objects and not-updated ones. Our model does not yield such problems, as the controller embeds a single law object, and the same law resides across all the controllers. Second, our model assumes that a law retains control over its own update procedure, thus defining and maintaining state consistency. Their model assumes an abstract, and unspecified consistent checkpoint for performing an update. Such a checkpoint can introduce severe limitations with respect to the types and the implementation of the objects that can be updated. Third, their model suffers from an unnecessary centralization for both providing an update, and for coordinating the update schedule throughout the system. Our model does not rely on a centralized update database, and can use sophisticated controller-to-controller communication for both propagating the update and for scheduling it at individual components. Last, our mechanism provides advanced access control to ensure that only proper

updates take place, where a proper update can be defined according to multiple and flexible criteria. Their model relies on a centralized entity to decide unilaterally what a proper update is, without employing additional local information useful in such a decision.

In addition to the research in the general area of software updates, there is a number of other works that address specific aspects of policy updates. First, Ray et al.[55, 54] address the issue of maintaining the consistency of a centralized system when updating its policy. The authors consider the effects of replacing an access control policy over the execution of transactions, and propose the abortion of those transactions that are affected by the change of policy. Furthermore, the change of policy is itself treated as an atomic transaction. The authors mainly use static methods for minimizing the number of aborted transactions taking into account both the old and the new policy as well as the type of transactions occurring when the hot update takes place. This objective is addressed in our work by: 1) the ability of the law to decide how, and when to perform the hot update in order to impact the system as little as possible, and 2) the lawChanged event and the compensative mechanism that can be employed in order to solve the conflicts dynamically, after the update has taken place. Unlike our work, this work is performed in an entirely centralized environment, where the policy does not have to be deployed throughout a distributed community. Moreover, the policies that are considered are of a very simple form, thus lending themselves to static analysis.

In a distributed environment, the authors of [24] address the issue of hot updating policies in the context of Ponder [23]. Ponder represents a language for specifying management and security policies for distributed system. Similar to LGI, Ponder enforces policies with respect to both the target (i.e, the resource), as well as the subject of an interaction. The enforcement is carried out by access controllers and by personal management agents. Policies in Ponder are represented by instance objects, which are central entities that handle an entire set of targets that are subject to that policy. A policy object maintains references to all the distributed policy enforcers. The main focus of the Ponder policy update is the dissemination of policies in the system. The lifecycle of a policy consists of enabling, disabling, unloading and deleting a policy. Policy updates are performed by successively disabling a policy and enabling a new one. The process is centralized, since it is directed by the policy object; moreover, the target or subject set, equivalent to the membership concept used in our work, is assumed to be available

through some type of central repository. The update process is carried out atomically for the entire policy set: a policy is disabled with respect to all agents in the system then a new policy is deployed for a given set. Although this approach eliminates inconsistencies due to partial updates, it can possibly produce large down times, where no policy is active in the system.

Drama [16] represents a policy-based network management system, designed to manage mobile ad-hoc networks. The policies are represented through a language that specifies high-level network requirements, such as quality of service, security, and management, as well as monitoring, filtering and reporting of data related to the network status. Policies are enforced through distributed Policy Agents, which are co-located with the network elements that are managed. Policy Agents are organized in a three-level hierarchy: A Global Policy Agent (GPA), a number of Domain Policy Agents (DPA), and the Local Policy Agents (LPA). A policy is disseminated from a GPA, through a DPA and down to the LPAs. The system is able to cope with disconnections between GPA and DPA, as well as between DPA and LPA. Similar to our distributed propagation of laws, the distribution of policies in Drama is not atomic: in the absence of communication means, an LPA can operate under the existing policy. As soon as communication is established with a Domain Policy Agent, an LPA can find what is the current version of the policy, thus it can request a newer policy, similar to our centralized pull update. The authors ignore inconsistencies that might appear due to different policies active at different agents, mostly due to the stateless character of the policies, as well as due to the more relaxed time constrains of a mobile environment.

Last but not least, XACML [30] is a standard language for specifying schemas for authorization policies, authorization decision requests and responses, as well as a model for evaluating such policies. In XACML, the policies are enforced through a hybrid scheme that involves distributed Policy Enforcement Points (PEP), co-located with protected resources, as well as a central Policy Decision Point (PDP), responsible for evaluating policies that apply to a given access request. The PDP obtains policies from a Policy Administration Point (PAP), or from a Policy Repository (PR), where a PAP stores its policies. Due to the centralized nature of the PDP, a policy update takes place atomically with respect to the entire system, by simply replacing the policy in the PAP and PR. The new policy will become effective starting with the next request served by the PDP. Access control decisions, however, are based also on attributes

associated with individual requests, somewhat similar to our concept of control state. Thus, the update process might provoke inconsistencies between the active policies and the used attributes. Presently, these inconsistencies can be resolved by separately and manually modifying the attributes using an Attribute Authority (AA) or an Attribute Repository (AA). As far as we know, there is no formal or automatic method to deal with such inconsistencies.

4.6 Summary

In this chapter, we have presented Hot Updates, a flexible mechanism that provides the update of laws in LGI. The updating model introduces primitives for promoting the hot updates at both individual agent, and community (system-wide) level; it also provides support for resolving inconsistencies that appear when different components are subject to different laws. The propagation of the updates throughout the system is not bound to a specific mechanism. Instead, Hot Policy Updates maintains its flexibility by providing the basic mechanisms for detecting mismatch of law version, leaving to the policy itself the freedom to use the most appropriate propagation method. We have shown a number of methods of propagation and we have discussed their advantages and drawbacks relative to the application type and the access control requirements.

Chapter 5

The Java-based Law Language

This chapter describes Java Laws, a novel Java-based language for expressing LGI laws, and the mechanism that supports it. In addition to providing basic law-writing capabilities, Java Laws is used as the underlying technology for implementing both Regulated Synchronous Communication and the Hot Updates. The most important features of Java Laws are the efficiency of the access control decision, the granularity of the evaluation, as well as the portability of the controller interpreting them. We start the chapter with a discussion of the advantages of Java Laws. We continue with a description of the structure of a law, the regulated events, and the primitive operations, in the context of both message-passing as well as synchronous communication. After presenting an example, we will continue with a more detailed discussion of the workspace of a law with emphasis on control state manipulation and access. We conclude with a section detailing the performance evaluation of Java Laws.

5.1 Motivation

When evaluating an access control mechanism, one has to take into account a number of pragmatic reasons, such as the performance it exhibits for various classes of applications, the ease of deployment, and the portability and suitability it manifests in different environments. Java Laws represents a novel Java-based language for expressing access control policies designed to achieve these goals, for LGI in general, and for Regulated Synchronous Communication in particular.

Law-Governed Interaction has been originally proposed in [43], based on the previous concept of “law-governed system”[42, 44], and it has been subsequently implemented in [46],[48], and [47]. Since inception, the original law language of LGI has been Prolog-based. Beside legacy arguments, the Prolog-based language has offered a number of advantages, such as a

declarative manner and a compact representation. However, the enforcement of such laws has been performed by the LGI controller, representing a proxy component written in Java. This combination—while preserving the advantages of Java in managing the communication and of Prolog in expressing policies—has been prone to large overheads, portability constraints, and other limitations.

Java Laws has emerged in this context; its declared purpose was to increase the performance and improve the portability of the controller, and eliminate the limitations manifested by the Prolog-based language. The use of Java-based language as a method of writing laws can be beneficial for the following pragmatic reasons:

- Increasing performance and portability: since the controller itself is written in Java, a policy that can be interpreted by a standard Java Virtual Machine will lead to better integration, decreased overhead in evaluating the policy, as well as increased portability inherent to the Java platform.

- Suitability and expressiveness: Java represents a language employed extensively in programming networked and distributed systems. Often, the communication in such systems employs the exchange of Java objects, as in the case of Remote Method Invocation (RMI) or Java Messaging System (JMS). The ability to explore and understand such objects improves the expressiveness and the granularity of the access control decision. Additionally, a Java based language can use a variety of available tools for interpreting and analyzing the data exchanged during interaction.

- Robustness: Java is a strong-typed language. By compiling a Java code, the code becomes less error-prone. Although a Prolog compiler/code-verifier can be devised, our experience show that it is relatively easy to make errors in a Prolog program; the discovery of such errors at runtime is a very laborious activity.

- Popularity: Java is a more popular language. It is beneficial to write laws in a popular, well-documented and widely-known language.

5.2 The structure of a Java-based law

Recall the definition of a law as a function `L(e, s)`, which returns a list of primitive operations, called the ruling of the law, for any possible regulated event `e`, and for any possible control state `s`. Java Laws represents a language based on Java for expressing such a function.

More concretely, a law written in this language, called a Java law, has two parts: a preamble and a body. The preamble contains a sequence of clauses reflecting the general characteristics of the law, such as: the name and identification of the law; a set of certifying authorities recognized by this law; identifiers for other laws required for inter-operability, etc. The body of the law is essentially a Java class, also called a law class, which extends a specific class, called the law base-class.

Every law class provides a set of event-methods, which are invoked by the controller whenever a corresponding regulated event occurs at it. The exact signature of these methods is specified in [62]. For example, the signature of the event method corresponding to the sending of a message is:

```
sent(String src,Datatype message,String dst,String dstlaw)
```

Each such method has access to the control state of the agent at hand, and is responsible for the evaluation of a ruling for the event that caused its invocation. The law class can also define any number of helper methods, designed to be called by the event methods.

5.2.1 The source code of a law

Figure 5.1 presents an example of a Java law, as written by a policy designer, and as submitted to a controller for enforcement. The law declares first the preamble followed by the body. In this example, the preamble consists of a single clause `law(L0,language(java))`, which identifies the name of the law (i.e. `L0`), and its type(i.e. java). The body of the law contains the definition of a law class, along with import clauses as they may appear in an ordinary file defining a Java class, per Java Language Specifications [31]. The class contains two types of methods: event methods and helper methods. By convention, the event-methods defined in the body of the law precede all helper methods. The example features two event-methods used in

```
/* Preamble of the law*/
law(L0, language(java))

/* The body of the law*/
import java.utils.*;
public class L0 extends Law {

/* Event methods*/
   public void sent(String src, Message msg, String dst, String destLaw) {
      doForward();
   }
   public void arrived(String src, String srcLaw, Message msg,String dst) {
      doDeliver();
   }

/* Helper methods*/
   ...
}
```

Figure 5.1: A simple Java law

the regulation of the message-passing communication: one deals with all sent events, and the other with all arrived events. The example shows no helper method.

5.2.2 The law class

The body of a Java law defines a law class. Unlike an ordinary Java class, a law class is restricted in several ways, and suffers pre-parsing and certain modifications when it is loaded in the controller. Most importantly, a law class always extends a specific class, called the law base class. In the case of message-passing the law base class is `moses.controller.Law`; in the case of RRMI, the law base class is `moses.controller.RMILaw` which in turns extends `moses.controller.Law`. A detailed description and the API of the base law class can be found at [62].

The role of the law base-class is to define events-methods corresponding to all possible regulated events. These methods, called base event methods, are implemented with an empty body, thus producing no results when directly invoked. A law class can provide event methods

```
/* Preamble of the law*/
law(RRMI-L0, language(java))

/* The body of the law*/
import java.utils.*;
public class RRMI-L0 extends RMILaw {

/* Event methods*/
   public void sent_rmi_call(String src, MethodCall mc, String dst, String dstLaw) {
      doForwardRmi();
   }
   public void arrived_rmi_call(String src, String srcLaw, MethodCall mc, String dst) {
      doDeliverRmi();
   }
   public void sent_rmi_result(String src, MethodCall mc, String dst, String dstLaw) {
      doForwardRmi();
   }
   public void arrived_rmi_result(String src, String srcLaw, MethodCall mc, String dst) {
      doDeliverRmi();
   }

/* Helper methods*/
   ...
}
```

Figure 5.2: A simple Java law for RRMI

that override the base event methods, thus offering a non-empty implementation.

The event-methods defined in a law class are called defined event-methods; in order to override the base events methods, they should follow a well defined signature. The role of the event methods is to compute the ruling of the law for the corresponding event in the context of the control state of the agent subject to this law.

Generally, the law base class defines a method for every event type. Whenever an event occurs, the method corresponding to this event is invoked. If the law class overrides this event method, then the overriding method is executed. In some cases, however, a law class may have multiple overloaded event-methods defined for a given event type. This is the case for events that handle messages. For convenience, the base law provides multiple event methods that are overloaded for different data types that can be carried in the message. The supported data types for messages are `String`, `Bytearray`, or `Object`, as well as a generic `Message` data type that acts as a fall-back method. The `Message` data type, and its use are discussed in Appendix C.

Thus, for every such message-related event, several event-methods might be defined. The event method to be invoked in such a case is chosen as follows: (a) if there is a defined event method that matches the data type of the payload exactly, then this event method is invoked; (b) if no such method is defined, then the event method with the `Message` data type argument is invoked; and (c), if conditions a) and b) are not satisfied, then the base event method is invoked, producing an empty ruling. The law defined in Figure 5.1 has two defined event-methods, dealing with sent and arrived events, respectively. The `message` argument in both methods is of type `Message`. As a consequence, this law allows for the non-obtrusive exchange of messages of arbitrary types. Prolog laws allow only the exchange of text messages, and only of a certain specific syntax.

RMI laws do not provide overloading of the base event methods, as various data types are accommodated in a single signature method. An example of simple, idempotent, Java law for Regulated Synchronous Communication is depicted in Figure 5.2. This example depicts the four specific event methods `sent_rmi_call`, `arrived_rmi_call`, `sent_rmi_result`, `arrived_rmi_result`. These methods provide access to the communication through the `Me-thodCall` object, that stores the parameters and identifiers of each method call, and

allows an easy access to their data. A detailed description of this object is provided in Appendix D.

5.2.3 The workspace of the law class

The workspace of a law class, representing classes, methods, and variables that are available to the law, can be classified into the following categories: (a) classes and methods that can be called in a law class, (b) member variables declared by the law class itself, and (c) the context variables provided by the base law class, and accessible in the law, including the control state of the agent. They are discussed below in this order.

The Methods and Classes Accessible to a Law-Class: These methods and classes fall into the following categories:

- Methods defined in the base law class are visible to all law-classes, by inheritance. These include the primitive operations, implemented as `doOp` methods, that contribute to the ruling of the law.
- Implicitly imported classes of the `moses` java package implementing the controller, like Term, Message, and some others. These classes are made available to every law class by automatic insertion of the corresponding import clauses in their text when the Java law is pre-processed by the controller. The reason for the automatic insertion is that these classes are commonly used in Java laws.
- Explicitly imported Java classes. In order to use classes other than the implicitly imported classes, a law class should explicitly import the desired packages. The classes allowed to be imported are classes available in the `CLASSPATH` of the controller interpreting the law; an example of such explicitly imported classes is the `import java.utils.*` clause shown in law L0. The law class cannot load/install by itself packages and classes other than those available to the controller itself.

The member variables declared in a law class: These variable must be final and static in order to prevent improper state transfer between different invocations of of event methods. Recall that in LGI, by definition, the ruling of an event only depends on the event itself and on the control state of the agent on whose behalf the ruling takes place. Thus, by restricting the use of non-final variables, we prevent the evaluation of different event methods effect each other in any way except via the official control state of the agent in question.

The context variables: Every law class inherits from its base class a number of context variables, such as the `Peer`, `PeerLaw`, `ThisLaw`, etc. These variables are initialized prior to evaluating any given event, and are available for direct access, as public member fields of the base law class. The most important of these variables, however, are `CS` and `DCS` , representing the law-based control state, and the distinguished control state, respectively. Access to these state variables is discussed below.

5.2.4 Access to the control state and to the ruling

Recall that the purpose of an LGI law is to compute a ruling, consisting of a list of primitive operations. In order to do this, the law class must have the means for examining the control state of the agent at hand and for contributing to this ruling. These means are discussed in this section.

Access to the control state of an agent at hand: The control state is exposed to the code of a law-class through two context variables, namely `CS` and `DCS`, representing the proper control state and the distinguished control state respectively. Semantically, both CS and DCS are represented, for legacy reasons, as bags (or lists) of Prolog-like terms that allow duplicates. Even though such terms are more natural for Prolog than for Java, we have chosen it for its clear model, as well as for interoperability with communities operating under Prolog laws. The Java Laws implementation of these terms is done by the class `moses.controlState.Term`, which is discussed in detail later in this section. Instances of this class are called “terms”, some of which may represent actually lists of terms, like the CS itself. The `Term` class provides methods to match terms based on various patterns, and retrieve their contents. The following

are simple examples of the use of some of these methods, when applied to CS.

- `CS.has("role(mgr)")` returns true, if the term `role(mgr)` has been found in CS.
- `CS.findT("level(%A)")` searches through CS for a term with the pattern `level(A)`, where "A" stands for any sub-term; the function returns the found term; the % character signifies an unbound variable, or a named wild card,).

Other methods for accessing lists of terms, and for analyzing terms are introduced later in this section. Finally, the representation of the control state as a member variable allows the code of a law class to update its local copy. However, such an update would have no effect on the state as seen by the subsequent evaluations. The control state can be updated in a persistent manner only by adding update operations, such as `add` and `replace` to the ruling of the law.

The computation of the ruling: For every primitive operation `Op(P)`, where `P` represents the arguments of operation `Op`, the base law class defines a method `doOp(P)`. Whenever such method is invoked, it would add the operation `Op(P)` to the ruling of the law. For example, the call `doAdd("level(0)")` would cause the operation `Add("level(0)")` to be appended to the ruling of the law. The calling of a `doOp` method does not execute the operation `Op` itself. The actual execution of this operation would be carried out by the controller, along with other operations in the ruling, after the evaluation of the law is completed.

5.2.5 Debugging and testing of Java laws

A simple and effective tool is provided by LGI in order to support the testing of laws in Java as well as Prolog. This tool provides with syntactic and semantic testing of laws and is presented in detail in [63]. In order to provide debugging of the runtime interaction, the developer of a law written in Java Laws can use the following mechanism. The developer of the law that requires debugging can incorporate printing statements (System.out.println(...)) in the code of the law. These statements will display the corresponding information at the standard output of the controller. This method provides more selectivity than the Prolog debugger regarding the information that is to be printed out during the debugging process.

5.2.6 Security-related limitations

During agent adoption or creation, a Java law is presented to a controller as a source file. The law is subsequently conditioned by separating the preamble and the body of the law. The body of the law is compiled just-on-time, and loaded. For security reasons, every Java law is loaded via a separate and individual class loader, such that its execution is sand-boxed. In order to insulate the execution of the law from the execution of the controller in the same JVM, the law follows a stricter model of the applet security. Consequently, a law is not allowed to access system resources, or resources on a different class loader, and to use the network and the file system. Indeed, a law-class is unable to access any variable or resource except those provided to by the law base class. Also, any execution of an event-method if it takes more than a time-limit imposed on it (this time-limit is a system parameter, currently set at a level of several seconds). Moreover, if a evaluation of an event method executes too long, and consumes too much memory, the evaluation is terminated and the result will be an empty ruling.

5.3 An example

In order to see how the ruling of the law is computed in Java Laws, let us consider the enforcement of the structure of a layered architecture. Layered architecture organizes large systems into disjoint groups of components, called layers. The components of a layered system are labeled with successive integer numbers, identifying the layer the component belongs to. The components are subject to the following global constraint:

1. Members of a layered system are assigned a non-negative integer that identifies their layer. The layer of a component is assigned dynamically, by a distinguished agent called `manager`, via a message `setLevel(k)` it sends to x.

2. Components can send messages to each other only if the sender resides at the layer of the target, or at the layer right above it; this constraint does not apply to the `setLevel` messages, mentioned above).

This law is enforced as follows. Every agent will carry its layer in its control state, as a term `layer(k)`. For every outgoing message, the law attaches the layer of the source to the

message. For every incoming message, the law will compare the layer attached to the message with the layer of the destination. If the two layers do not reflect an appropriate relation, the message is denied; otherwise the attached layer is stripped off the message, and the original message is handed down to the actor.

Figure 5.3 shows the implementation of the Java law for the layered architecture. The law contains three event methods, `adopted`, `sent`, and `arrived`, as well as a helper method, `getContentFromMessage` that is used in the `arrived` event method. The rest of this section describes how the layered architecture law operates.

Beside the name of the law, the preamble of the law declares a certifying authority, `sysAdmin`, identified by the hash of its public key. The certifying authority is used for authenticating the manager assigning the dynamic layers. In the `adopted` event method, every agent that starts operating in the community receives a term `layer(0)` in its control state, thus getting assigned into layer 0 of the system. Additionally, if the agent joining the community submits a certificate, and if the certificate is properly signed by the `sysAdmin` authority and contains the `role(manager)` attributes, then a `role(manager)` term is also added in the control state of the agent. This term represents a marker identifying the manager, and it will later allow the agent to assign the appropriate layer to other agents.

The `sent` event method captures all the messages emitted by an agent. If the message is of the form `setLevel`, it is allowed to pass only if the source is recognized as a manager by the marker term in the control state. Otherwise, if the message represents regular communication, it will be subject to the layered architecture restrictions. Thus, the level of the source is retrieved from the control state, and it is forwarded along with the message in a special `extendedMessage` envelope.

The `sent` event method deals with all the incoming messages. If the message is of a `setLevel` form, then the level is retrieved from the message, and the new level is replaced in the control state using the corresponding primitive operation method `doReplace`. Otherwise, if the message is an `extendedMessage`, then the layered in the control state is compared with the layered attached to the message. If they match, the message is stripped of the envelope and is handed down to the destination. This event method uses the helper method `getContentFromMessage` in order to retrieve the original message from the envelope. A similar

```
law(LayeredArchitecture, language(java))
authority(sysAdmin,keyHash(————))

public class LayeredArchitecture extends Law {
   public void adopted(String arg, String[] issuer, String[] subject, String[] attributes) {
      if (issuer.length != 0) {
         if (issuer[0].equals(”sysAdmin”) && subject[0].equals(Self) &&
               attributes[0].equals(”role(mgr)”)) {
            doAdd(”role(mgr)”);
         }
      }
      doAdd(”level(0)”);
   }
   public void sent(String source, String message, String dest) {
      if (message.startsWith(”setLevel”) && CS.has(”role(manager)”) ) {
         doForward();return;
      }
      int levelK = CS.fetchInt(”level”);
      doForward(source, ”extendedMessage(” + levelK + ”,” + message + ”)”, dest);
   }
   public void arrived(String source, String message, String dest) {
      String content = getContentFromMessage(message);
      Term levelTerm = CS.findT(”level(%A)”);
      if (message.startsWith(”setLevel”)) {
         doReplace(levelTerm.toString(),”level(” + content + ”)”);
         doDeliver();return;
      }
      if (message.startsWith(”extendedMessage”)) {
         String trueMessage = content.substring(content.indexOf(’,’) + 1);
         int levelK = Integer.parseInt(content.substring(0, content.indexOf(’,’)));
         int levelK1 = CS.fetchInt(”level”);
         if ((levelK >= levelK1) && (levelK <= levelK1 + 1)) {
            doDeliver(source, trueMessage, Self);return;
         }
      }
   }
   public String getContentFromMessage(String anyMessage) {
      int index = anyMessage.indexOf(”(”);
      if (index == -1) return ””;
      else return anyMessage.substring(index+1, anyMessage.length()-1);
   }
}
```

Figure 5.3: Java law for a layered system

logic is used for retrieving the source layer from the message, using ordinary Java code directly in the event method.

5.4 Control state access and manipulation

The control state and the distinguished control state are defined as bags (or "multi-lists") of Prolog-like terms. Terms can be defined, recursively, as follows: a term is either an atom s or the structure $f(t_1, ..., t_n)$, where f, called the functor, is an atom, and each t_i is either an atom or, recursively, a term. Here, an atom is either a short string like "john," or a number, such as "17" (See [20] for a more precise definition). The following are some examples of such terms: $manager$, $role(manager)$, $name(john, doe)$, and $name(first(joe), last(smith))$. There is one type of term that has the special form: $[t1, t2, ..., tn]$, which represents a list of n terms, for any n. Both the CS and the DCS are represented as such lists of arbitrary terms.

A Java Term object features methods that provide the following capabilities: (1) conversion between the string representation of terms, such as above, and their internal representation; (2) retrieving terms with specific structure, from a given list of terms, via pattern matching; (3) direct access to different components of a given term. The methods that provide these capabilities are described below. Appendix B presents the detailed structure of `Term` objects, and a number of low-level methods for direct term manipulation. The low-level methods, although more complex, deliver a better performance.

5.4.1 Conversion Between the String and Internal Representation of Terms

The static $parse$ method of class Term creates a term object from its String representation, as follows:

```
Term emp=Term.parse("emp(name(john,doe),roles([ceo,char]))");
```

The inverse conversion is carried out via the method $toString$. Thus, the following:

```
String s = emp.toString()
```

would store in s the string parsed above.

5.4.2 Unification of terms with Patterns

The means provided for the analysis of terms is via unification based pattern matching (see [20] concerning the concept of unification). A pattern is a term—in its String form—some of whose sub-terms are replaced with named variables, denoted by $\%V$, where V is any symbol. For example, the pattern:

```
"emp(name(%N,doe),roles(%R))"
```

would match the term emp defined above, binding variable $\%N$ to "john", and variable $\%R$ to the list "[ceo,chair]". The pattern variables serve as wild-cards, but the fact that they are named provides some important capabilities. First, they allow one to retrieve the values they have been bound to by the unification, as we shall see below. Second, they provide the ability to impose useful constraints on the unification by repeating the same variable in different places in the pattern. For example, the pattern

```
"emp(name(%N,%N),%R)"
```

would match an employee term only if its first and last names were identical.

5.4.3 Search Through Lists of Terms

Class Term provides several methods that when applied to a list of terms, such as CS or DCS—recall that a list of terms is itself a term, thus an instance of class Term—would scan this list attempting to find the first term in the list that matches the patterns given as an argument. In fact, these methods can be applied to non-list terms searching through their first-level sub-terms; however, this capability is not likely to be used often when writing laws. We will introduce these methods, applying them to the context variable CS—which represents their most common usage—and using $''p''$ to represent arbitrary patterns. The search methods of Term are as follows:

- The $CS.findT(p)$ method: This method returns the first element of the list-term it operates on, that matches pattern p; or, it returns null if no such term has been found. For example, the following:

```
Term emp = CS.findT("emp(name(%A,smith),roles(%R))")
```

  would return into emp the first employee term in CS whose last name is "smith", and whose first name, and roles, are arbitrary.

- The $CS.has(p)$ method: This is like the $findT$ method, except that it does not return the term it found, but a boolean value indicating whether or not it has been found.

- The $CS.fetchInt(p)$ Method: This is a specialization of the $findT$ method, which is expected to be used often. The pattern p in this case is stands for a functor, like "level". The call $CS.fetchInt("level")$ would search for a term of the form $level(I)$ where I is any integer, and returns that integer.

5.4.4 Analysis of a term picked up from a term-list

Class Term features another search method called find, which operates like $findT$, but it returns an object of a special type $UnifyResult$, which provides for the retrieval of the whole matched term, as well as, of the values bound to the variables of pattern p used to retrieve it. Consider, for example, the following statement:

```
UnifyResult emp=CS.find("emp(name(%A,smith),roles(%R))'')
```

We will see next how the various $UnifyResult$ methods to be introduced below operate on object emp, retrieving various parts of the found term.

- The $emp.getTVar(var)$ method: This method returns the value bound to variable "var" by the unification carried out via method find above. Specifically, the statement:

```
Term roles = emp.getTVar("Roles")
```

would return in the Term variable $roles$ the list of roles of the employee at hand.

- The $getSVar(var)$ method: This is like the method $getTVar$ above, except that it returns its result as a String. Specifically, the statement:

```
String sRoles = emp.getTVar("Roles")
```

would return in the String variable $sRoles$ the list of roles of the employee at hand.

- The $emp.getTerm()$ method: This method returns the entire term stored in emp—which would be the result of calling $CS.findT(...)$ method.

5.5 The Performance of Java Laws and of the Controller

In this section, we will present a number of results we have obtained in evaluating Java Laws. Since the performance of the Java Laws is inextricably related to that of the controller, some of the measurements we provide here incorporate the performance of both.

5.5.1 Java Laws Event Evaluation

In this set of experiments, we aimed to asses the evaluation time for Java Laws, as absolute value, as well as compared with event evaluation time for Prolog laws. In order to separate the law evaluation time from the communication time, we have used the following scenario. We have employed a single actor and a single controller located in the same LAN; the actor sends an LGI-message to itself. When this message reaches the controller, a specifically designed law is used to trigger a recuring event. After the sequence of recurring events has been evaluated, the message is delivered back to the actor. We have measured the time since the agent has issued the message until after it has received it. The measured time reflects a roundtrip communication time between the actor and its controller, plus the time to evaluate the law for a sequence of

events. When the sequence of events is large enough, the communication component becomes negligeable. Figure 5.4 displays the law used to generate the sequence of recurring events. The adopted event creates a counter `round(1000)` in the control state of the agent. When the agent sends a message (in `sent` event-method), the counter is decremented and the message is `forwarded` to itself. The forwarding of the message will cause an `arrived` event at the agent. When the arrived event is evaluated, the counter is decremented and the message is forwarded again to itself. This forwarding will cause, in turn, another arrived event at the agent. This recurring sequence of events is finally interrupted when the `round` counter reaches zero; in this case the message is delivered back to the actor. An equivalent Prolog law has been used for assessing the evalutaion time for Prolog laws.

```
/* Preamble of the law*/
law(Benchmark, language(java))

public class Benchmark extends Law {
   public void adopted(String args) {
      doAdd(”round(1000)”);
   }
   public void sent(String src, String msg, String dst, String destLaw) {
      doDecr(“round”,1); doForward(Self, msg, Self);
   }
   public void arrived(String src, String srcLaw, String msg,String dst) {
      int round = CS.fetchInt(“round”);
      if(round¡=0)
         doDeliver();
      else {
      doDecr(“round”,1); doForward(Self, msg, Self);
      }
   }
}
```

Figure 5.4: Evaluation speed benchmark law

The experiments have been conducted on various platforms, in different configurations, as presented in Figure 5.5. We have used a SUN JVM 1.4.0 and 1.5.2. Measurements have been averaged on 1000 internal events, as shown in Figure 5.4. An increase in the number of internal events did not show any improvement in performance.

Name	CPU Type	CPU speed	Memory	Operating System
ramses	SunUltra10	440Mhz	256M	Solaris SunOS5.8
ramses-pc	AMDK6-2	400Mhz	128M	WindowsNT4.0 Workstation
redwine	Intel686PIII	550Mhz	256M	Linux 1.6
h-pc	IntelP4	1.5Ghz	384M	Windows2000 Professional
mco	IntelP4Dual	3.2 GHz	1G	Linux 2.6.8.121

Figure 5.5: Platforms running the event evaluation speed experiment

name	Prolog	Java Laws
ramses	3.07185ms	1.00185ms
ramses-pc	-	3.8ms
redwine	-	578μs
h-pc	-	570 μs
mco	2ms	50 μs

Figure 5.6: Average event evaluation time

Table 5.6 shows the absolute values of the average evaluation time for both Prolog and Java Laws. It can be seen that Java Laws performs between 3 and 40 times better than the Prolog laws. Due to portability problems, we were unable to obtain results for Prolog laws in some of the platforms. The law used for the experiment features minimal computation during the evaluation of the events. Most laws that we have used so far, however, had shown minimal impact on the evaluation time.

In order to put the evaluation time in perspective, the communication time between two Java applications using TCP/IP is several orders of magnitude higher than this evaluation time. While this communication time might vary widely based on the hardware/software in use, typical values we have obtained are 1 ms for LAN communication using Ethernet 100 base-T and 100 ms for a WAN over 25 hops. This times represents end-to-end Java communication, including serialization for relativelly short messages, few hundreads of bytes in size.

A number of other experiments have been carried out in order to asses the robustness and scalability of a single controller employing Java Laws. These experiments had been conducted within a LAN, using our “mco” Linux workstation in Figure 5.5. Below, are brief reports of some of these experiments.

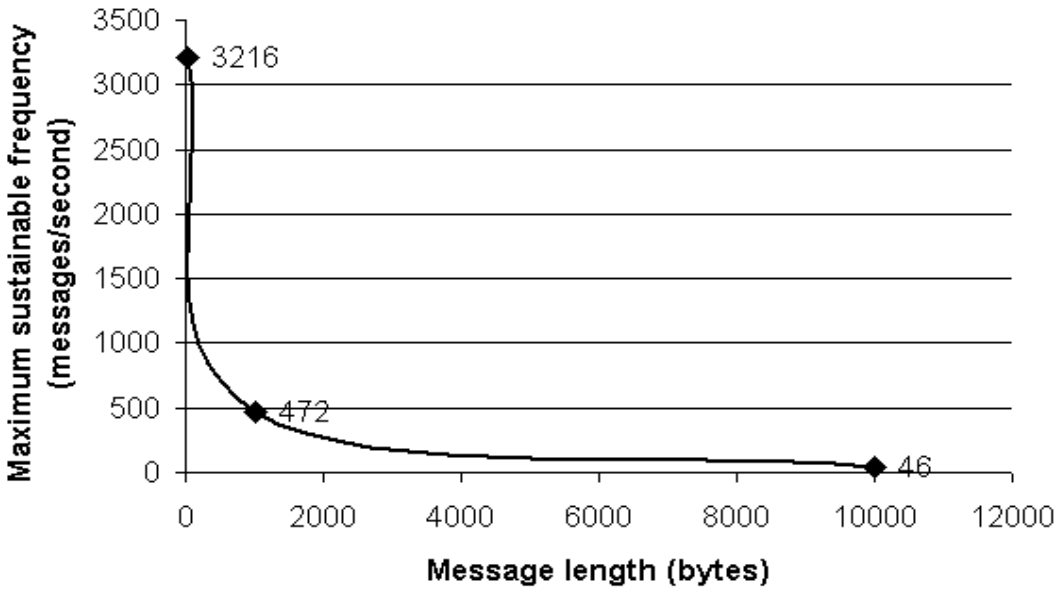

Figure 5.7: Maximum sustainable frequency

5.5.2 Maximum Sustainable Frequency

This experiment determines the maximum sustainable frequency of messages a controller can handle. In this setup, we have used two actors that adopted the same controller: the same controller thus operates two private-controllers. One actor sends messages to the other actor with a given frequency. The sustainable frequency is the frequency that the controller can sustain over long periods of time. The controller handles messages at this frequency without dropping, queuing up, or other errors. This frequency is measured over a long session of communication (20 minutes to 1 hour) (the burst frequency we experienced is much higher due to internal buffering and queuing in the controller.). The maximum sustainable frequency (messages/second) is measured for various message lengths (bytes), and the result is depicted in Figure 5.7:

5.5.3 Concurrent event evaluation

This experiment shows the performance of the controller in evaluating events when multiple agents are adopted by the same controller. Two parameters are measured: 1) the average evaluation time – representing the mean time it takes the controller to evaluate an event when other concurrent events are present; and 2) the controller throughput representing the number of events the controller handles (from multiple sources) within one second. Each parameter

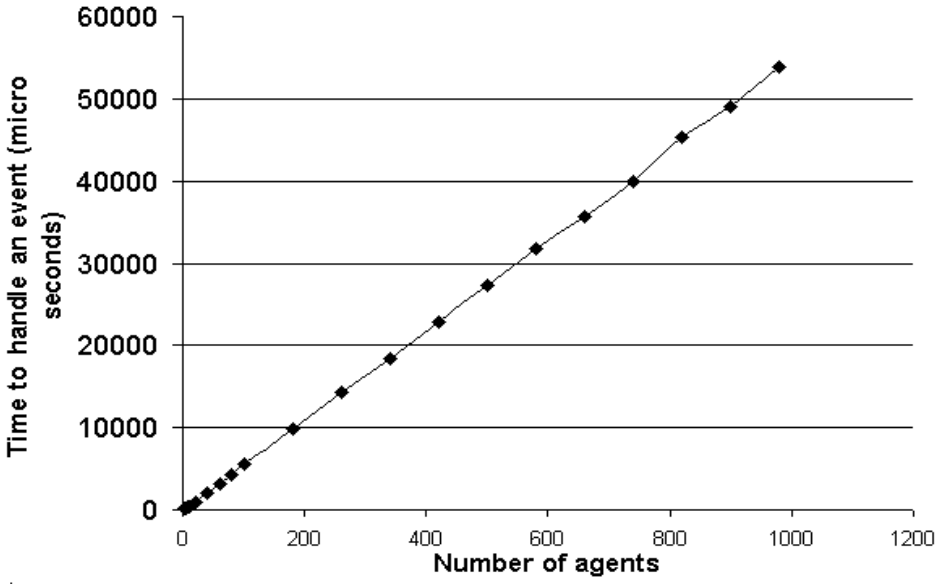

Figure 5.8: Average event evaluation time

is measured when the controller handles concurrently from 2 to 1000 agents. The events are law-generated using the same method presented in Figure 5.4, averaged over 50,000 events, and with messages of 20 bytes in size.

The results of this experiment show a linear increase of the evaluation time with the number of agents, as displayed in Figure 5.8. The evaluation time is proportional to the value of 50 micro seconds per event and per agent. The throughput of the controller is stable, at a level of 18500 events per second for a large number of agents, and slightly higher for a smaller number of agents, as seen in Figure 5.9.

5.5.4 Round-Trip Time

This experiment measures the performance of the controller in handling the communication between a pair of agents when multiple pairs of agents share the same controller. This experiment is intended to reproduce the performance in a real-life operational environment, where multiple agents communicate with the controller simultaneously.

Two parameters are measured. The first parameter is the average RTT, representing the mean time it takes an agent to send a message to its pair, through the controller and receive an answer back. This measurement is performed while other pairs of agents communicate simultaneously, and in a similar fashion. The second measurement represents the controller

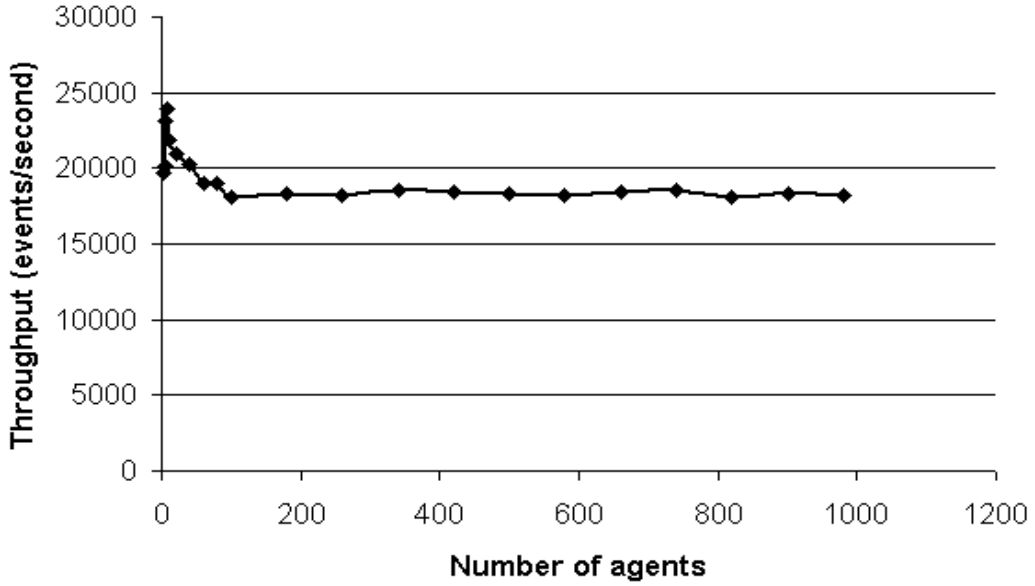

Figure 5.9: Controller throughput

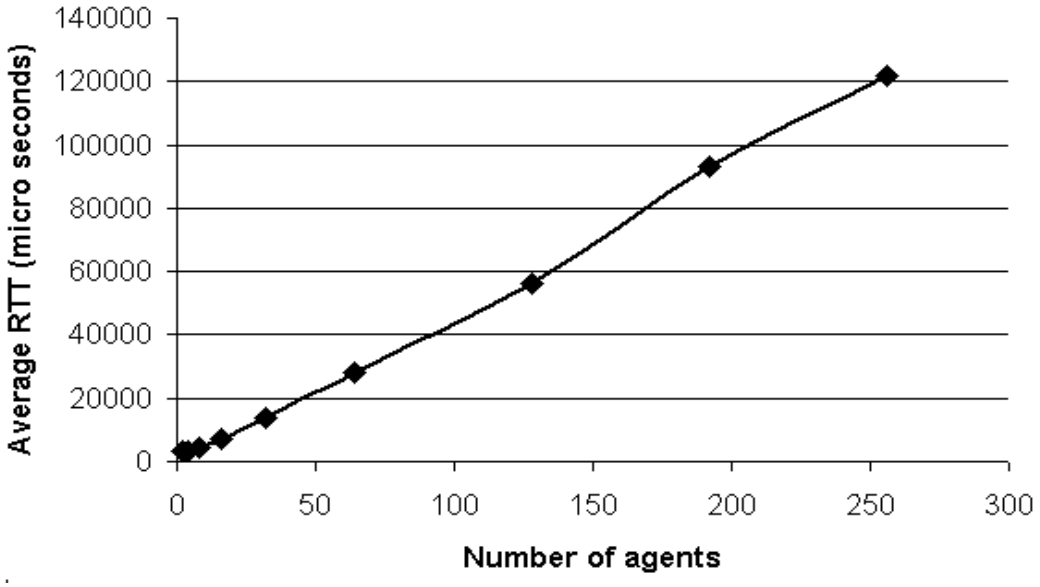

Figure 5.10: Average Round-Trip Time

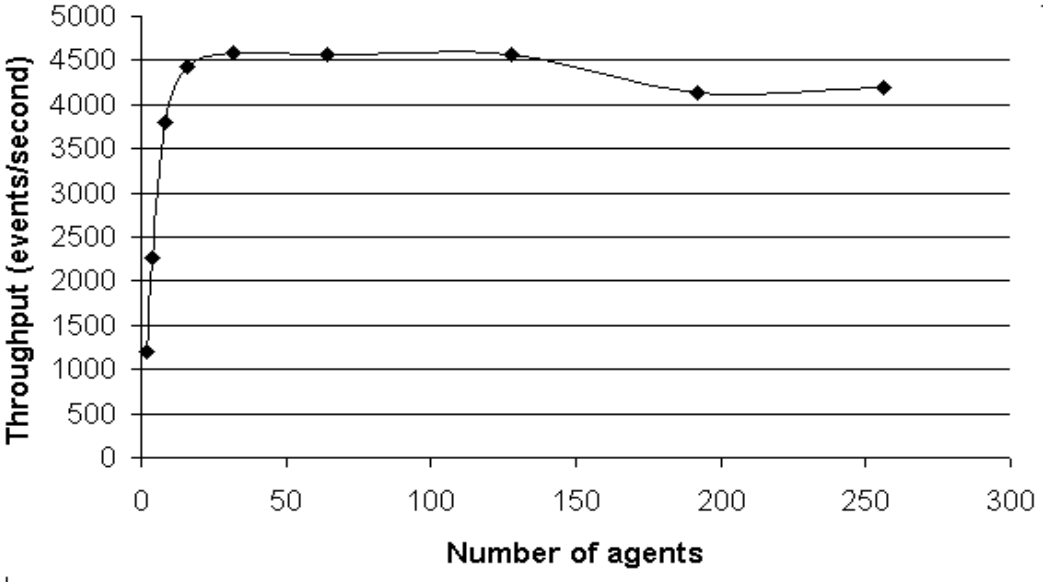

Figure 5.11: Controller throughput

throughput – the number of events the controller handles (from multiple sources) within one second. Each parameter is measured when the controller handles concurrently from 2 to 256 agents.

The RTT reflects the evaluation of 4 events by the controller as well as 4 corresponding LAN communication. The throughput of the controller is limited by the network and IO operations related to each message. The results presented in Figure 5.10 and Figure 5.11 show the performance of a controller interpreting a trivial Java law.

On average, when multiple agents communicate with the controller, it takes 220 micro seconds to receive (or to send a message) and to handle the associated event. This value sets a relatively stable throughput rate for the controller to an average of 4500 events per second.

5.5.5 Actor to Controller Communication

The following average values have been observed for end-to-end actor to controller communication when measured at the application level at the actor.

- When the actor and the controller are located on the same host, the message handling time is 130 micro seconds.
- When the actor and the controller are located on different hosts, on the same LAN, the

message handling time is 1 mili-second.

Both values above reflect the time to send a TCP/IP message (in the first case within the same host, in the second time across the LAN), as well as a single event evaluation at the controller. The controller was interpreting a trivial Java law.

5.6 Summary

In this chapter, we have presented Java Laws, the Java-based language for writing laws, and the mechanism that supports it. We have designed Java Laws for the purpose of: a) increasing the performance of the controller, b) improving its portability across multiple platforms and operating systems, c) gaining a better granularity in controlling various data types exchanged during communication, with emphasys on interaction within distributed Java applications, and d) increased robustness and scalability of the overall controller. The model we have presented in this chapter along with the performance evaluation prove that we have achieved these goals. We have also presented a number of example proving that the Java Laws language is simple and expressive for designing access control policies for distributed applications written in Java.

Chapter 6

Future Work

The research presented in this dissertation can be augmented in many promising ways. Below is an outline of certain specific directions that I consider important, and I plan to pursue in the future.

The main objective of this dissertation was to advance and develop the mechanisms of access control. The method we employed was to regulate the communication between all the distributed components of a system. But interaction between the components of a large system often takes place not only remotely, i.e., across the network, but also locally, i.e., within the same address space. For example, in the case of RMI, the client and the server exchange objects with each other, either as arguments, or as a result of a call. Although conceptually such objects might be considered part of the component that originally created them, they might end up being used in a different address space in the aftermath of the remote interaction. Other reasons for having distinct components co-located within the same address space are re-deployment of components, software evolution, or the dynamic-loading features of the system. In such conditions, it would be desirable to have the same comprehensive and sophisticated control policy applied to both local and remote interaction, in a unified manner. Among the major challenges facing such a mechanism are: a) how to define, or identify, what constitutes distinct components sharing the same address space, b) how to intercept the communication between such components in a manner similar to that employed in RRMI, thus controlling both the request and the reply, with maximum flexibility, and c) how to do this efficiently, i.e., with an acceptable overhead, given that local calls are typically many orders of magnitude more efficient than remote calls.

Another area where this work can be extended is the adaptation, or retrofitting, of legacy applications for enabling advanced regulation. Given the similarities between the source code

of an application developed using the RRMI suite and the source code of an application developed with the bare-bone Java RMI, the interchange between the two protocols can be performed in a straightforward manner. Indeed, in Section 3.4 we have presented an automated tool for performing such a replacement. But, in order to use such a method for large and complex software, it is necessary to take into account other aspects beside the swapping of the two protocols. First, a code previously not equipped for access control should be augmented with a certain security context in order to provide authentication, credentials, and other information relevant to the access control decision. Such information might not have a static character and it might have to be updated throughout the lifetime of an agent, and can be dependent on the interaction stage. Second, the code should be adapted in order to cope with the results of the access control decisions, including the denial of access, or the request for additional information or specific input from the agent. While applications developed with Java RMI are usually prepared to handle basic errors related to communication, handling security exceptions will require additional logic to be plugged in. And third, associating an agent to an address space—as we have provided—might prove to be too coarse, and might not reflect all the interactions taking place in a distributed system. A finer granularity might be necessary, where different RMI calls originating in the same address space can be associated with different principals, thus enabling different access control policies to guide distinct flows of interactions.

The access control mechanism proposed in this dissertation improves the security of a distributed system by preventing malicious agents from using the system in an unintended way. The same mechanism, however, can be employed for increasing the dependability of a distributed system in front of faulty components, instead of against malicious components. While the capabilities of the present mechanism provide sophisticated means for detecting faulty components or behaviors, the methods usually employed in handling the faults are significantly more complex than those of access control. Beside detecting faults, the mechanism we have developed must also employ sound error recovery schemes designed to enable the system to cope with such faults. The conventional model for error recovery in distributed systems relies on checkpointing and rollback recovery techniques. While this is an appropriate method in certain cases, forward recovery methods are often necessary in order to achieve application progress. Forward error recovery has been applied successfully in centralized systems,

most often through exception mechanisms in programming languages. In large and distributed systems, however, the forward error recovery often requires the cooperation of multiple components when handling exceptional situations. Such cooperation has to be precisely coordinated using a global framework that is yet to be designed.

Chapter 7

Conclusions

In the modern age, society as well as individuals increasingly depend on advanced computing applications for performing day-to-day activities and for satisfying their communication needs. Such critical applications often consist of great numbers of components, heterogeneous in nature and distributed on a large scale across the Internet. The critical nature of the applications on the one hand, and their potential vulnerabilities on the other hand, require advanced access control mechanisms to protect them against both malicious attacks and unintentional abuse. Accordingly, modern access control demands an advanced degree of expressiveness in order to capture the semantics and the details of various interactions. It also requires ability to enforce policies globally across all the components of the application, in a distributed and scalable manner, thus suitable for sufficiently large applications.

Law-Governed Interaction (LGI) represents a coordination and control mechanism that has been previously shown to satisfy these demands. LGI allows a group of distributed actors to engage in a mode of interaction governed by an explicitly specified and strictly enforced policy, called the law of the group. LGI, however, has been previously defined for asynchronous, message passing, communication, leaving unsupported the wide range of applications that employ synchronous communication. Furthermore, no formal mechanism had been designed for updating its policies when such action is deemed necessary—an aspect of great importance to any policy-based mechanism.

This thesis provided answers to the following questions:

- How to perform access control for synchronous communication? How is it different from access control for message passing, especially in the context of expressive and overarching policies?

- How to change the policy of a distributed system in an on-line manner when the policy

itself is distributed, as in the case of LGI? How to minimize the impact of such changes on both the system and on the access control itself?

This dissertation presented **Regulated Synchronous Communication**, an extension of LGI designed to provide advanced access control for synchronous communication, and **Hot Updates**, a model for updating the laws of a system.

To the best of our knowledge, Regulated Synchronous Communication represents the first access control model that takes into account the innate properties of synchronous communication. The most notable characteristics of the resulting regulation model are: (a) control of both the request part and the reply part of a call; (b) regulation performed both at the client and at the server side; and (c) control of the timing of the interaction and explicit timeouts handled in a manner that can take into account the concerns of both the client and the server. While the full power of the proposed mechanism resides in its ability to handle stateful and communal policies, we believe that this model is useful for access control in general, under less sophisticated requirements. Its implementation, Regulated RMI, can also be used for the customization of synchronous protocols even when the access control is not necessary.

Hot Updates represents a model for changing the policy of the system with a minimal impact on the operation of both the system and its policies. Hot Updates addresses a number of novel issues, such as: how to propagate the policy updates throughout the system; when to update the policy with respect to an individual component; and how to avoid, minimize or compensate possible inconsistencies that might appear during the update process, both at the component and at a system-wide level. According to this model, the update process itself is subject to the policy: the policy controls how the process is initiated and how updates are propagated. Additionally, Hot Updates provides support for resolving inconsistencies that appear when different components are subject to different versions of an access control policy. Hot Updates maintains its flexibility by providing various methods for propagating the updates, suitable for a wide range of systems and access control policies.

This dissertation also introduced Java Laws, a Java-based language for LGI laws, that has been employed in implementing both Remote Synchronous Communication and Hot Updates. Java Laws provides an expressive language for representing LGI laws, and an efficient evaluation mechanism seamlessly integrated within the LGI controller. This integration offers

portability by enabling the deployment of the controller infrastructure across various operating systems and platforms. Java Laws also provides a common platform for applying fine-grained access control particularly suitable for distributed applications written in Java. For such applications, Java Laws provides the mechanism to interpret the pending data exchanges, enabling access to in-transit arguments or replies represented as Java objects, thus leveraging the understanding of the occurring interaction, a necessary aspect of a sophisticated access control decision. This feature is particularly useful when applied to systems developed using Java RMI technology, effectively providing the basis for implementing the Regulated RMI suite. The experimental results we have provided for both Regulated RMI as well as for LGI message-exchange show that Java Laws provides a very efficient evaluation mechanism, and introduces a low overhead relative to the end-to-end communication.

Therefore, the security requirements of modern large scale applications can be addressed using an efficient and sophisticated access control model that takes into account the specific mode of communication employed by the system, and the ever changing character of the access control policies.

Appendix A

Hot Updates Primitives

This section presents the comprehensive list of LGI primitives designed to offer support for the Hot Updates. The new primitives are implemented using the Java Laws language. At the time of writing this document, these primitives were not yet ported to the Prolog-based law language of LGI.

```
previous(URL)
```

represents a preamble clause that declares a law to be part of a given lineage. `URL` represents the actual URL where the parent law can be downloaded from.

```
public void doUpdate(String url)
```

represents the primitive operation that triggers an update of a law. `URL` represents the actual URL where the new law can be downloaded from.

```
public void lawChanged(String newlaw, String oldlaw)
```

represents the first event on behalf of the agent after its law has been updated. `newlaw` represents the name of the current (i.e. new) law as it appears in the `Law` preamble definition of the new law; `oldlaw` represents the name of the parent law as it appears in the `Law` preamble definition of the parent law.

```
public void ghost(int etype, Object payload)
```

represents the `ghost` regulated event triggered in a law as a consequence of detecting an event designed to be handled under the previous law. `etype` represents the `ID` of the event, as defined in `moses.util.Const` class, and `payload` represents the actual event payload. The following are the possible values of the `etype` fields, their significance as well as the corresponding data type for the `payload` object.

- int OBLIGATION_T_E = 0 - obligation timeout event;
  payload type: `moses.controller.Obligation`
- int STATE_OBLIGATION_E = 1 - obligation state change event;
  payload type: `moses.controller.StateEvent`
- int ARRIVED_I_E = 2 - arrived internal event;
  payload type: `moses.message.Message`
- int ARRIVED_E_E = 3 - arrived external event (imported);
  payload type: `moses.message.Message`
- int SENT_I_E = 4 - sent internal event;
  payload type: `moses.message.Message`
- int SENT_E_E = 5; - sent external event (export);
  payload type: `moses.message.Message`
- int CERTIFIED_E = 6 - certified event;
  payload type: `moses.controller.CertVerifier`
- int ADOPTED_E = 7 - adopted event;
  payload type: `moses.controller.CertVerifierList`
- int CREATED_E = 8 - created event;
  payload type: `moses.message.Message`
- int EXCEPTION_F_E = 9 - exception-forward event;
  payload type: `moses.message.Message`

- int EXCEPTION_C_E = 10 - exception-created event;
  payload type: `moses.message.Message`
- int EXCEPTION_D_E = 11 - exception-deliver event;
  payload type: `moses.message.Message`
- int EXCEPTION_R_E = 12 - exception-release event;
  payload type: `moses.message.Message`
- int SUBMITTED_E = 14 - submitted event;
  payload type: `moses.message.Message`
- int SUBMITTED_P_E = 15 - submitted-pwd event;
  payload type: `moses.message.Message`
- int SUBMITTED_C_E = 16 - submitted-certificate event;
  payload type: `moses.controller.CertVerifier`
- int DISCONNECTED_E = 17 - disconnected event;
  payload type: null
- int RECONNECTED_E = 18 - reconnected event;
  payload type: null

```
public void exception(Message m, String failurecause)
```

represents a modification to the exception event. The enhancement refers to the `failure-cause` argument taking a new value, destinationLawObsolete, when the intended destination law is a newer version of the actual destination law. This is the case when the destination has not yet updated its law, but the source assumes that the destination has already done so. The behavior of the event and the `m` argument maintain the same semantics as previously defined in LGI. The `destinationLawMismatch` failure cause that previously covered this case has thus been restricted, and it does not covered the mismatch between different versions of the same law.

Appendix B

Java Laws: The Structure of the Term Objects, and Low-level Term Operations

The low-level operations on terms allow manipulation and construction of terms while exposing the implementation details of the `Term` class. The low-level implementation of the Term class has been directed towards efficiency and simplicity of use. Due to this reasons, the class does not provide with checks for illegal operations: special attention should be paid while working with the low level operations in order not to leave the Term object in an inconsistent state. Following is the list of fields of the Term object and their description:

- `int type` represents the type of this term. The type of the term directs what are the expected values of the other field variables for this object. The type can have any of the following values: `Term.SType` (defined as 0), `Term.IType` (defined as 1), `Term.FType` (defined as 2), `Term.LType` (defined as 3), `Term.CType` (defined as 4).

- `String functor` In the case of an atomic string term (`type == Term.SType`), this field holds the value of the atom. In the case of compound term (`type == Term.CType`) this field holds the functor of the term. In the case of the other types of terms, this field is ignored.

- `int IValue` In the case of an atomic integer term (`type == Term.IType`), this field holds the value of the atom. In all other cases this field is ignored.

- `float FValue` In the case of an atomic float term (`type == Term.FType`), this field holds the value of the atom. In all other cases this field is ignored.

- `Vector VValue` In the case of compound terms or lists (`type == Term.CType || type == Term.LType`), this vector holds all the subterms of this term. In order for

a term to be properly formed, the vector should be allocated to the proper size, and it should contain ONLY Term objects. Note that a list can have the arity zero (thus this vector size could be zero), while a compound term could not.

In order to create a Term object, the following constructors are provided:

- `Term(String svalue)`: Creates an atomic string term (`type = Term.SType`), and initializes its `functor` field with the value in the argument.
- `Term(int ivalue)`: Creates an atomic integer term (`type = Term.IType`), and initializes its `IValue` field with the value in the argument.
- `Term(float fvalue)`: Creates an atomic float term (`type = Term.FType`), and initializes its `FValue` field with the value in the argument.
- `Term(double dvalue)`: The same as above, except that the float value is converted from the double argument.
- `Term(String functor, int type)`: This constructor creates a term given its type. If the type argument is `Term.SType`, then it creates an atomic string term initialized with the functor argument. If the type is `Term.IType` or `Term.FType`, then the functor is parsed to the appropriate data type, then stored in the corresponding field. If the type argument is `Term.CType` or `Term.LType`, the object type is set accordingly, the functor is set to the argument value, and the VValue vector field is initialized.

Beside direct access to the field variables (all fields are public), an additional number of methods are provided:

- `int getArity()` returns the arity of this term for compound or list terms. This method simply returns the size of the `VValue` vector. If this field is not initialized (as in the case of e.g. IType) this method throws an exception.
- `Term get(int index)` returns the subterm with the given index that belongs to this term. This method indexes the `VValue` vector, with no preliminary check. If this field is not initialized (as in the case of e.g. IType) this method throws an exception.

- Term addST(Term term) adds the subterm argument to this term. It returns this object.
- `String toString()` returns a String representation of this Term. It is the inverse operation of `Term.parse(String term)`.
- `boolean deep_equals(Object obj)` represents an implementation of the equals method for terms. In order for two terms to be equal, they should have the same type and their corresponding field should be equal. In the case of compound or list terms, every component of the vector should be equal to the corresponding component of the argument.
- `Term deep_clone()` creates a deep clone of this object. The fields of the new object will have their own variables. In the case of compound or list terms the vector field will hold a clone of all the component in the original object.

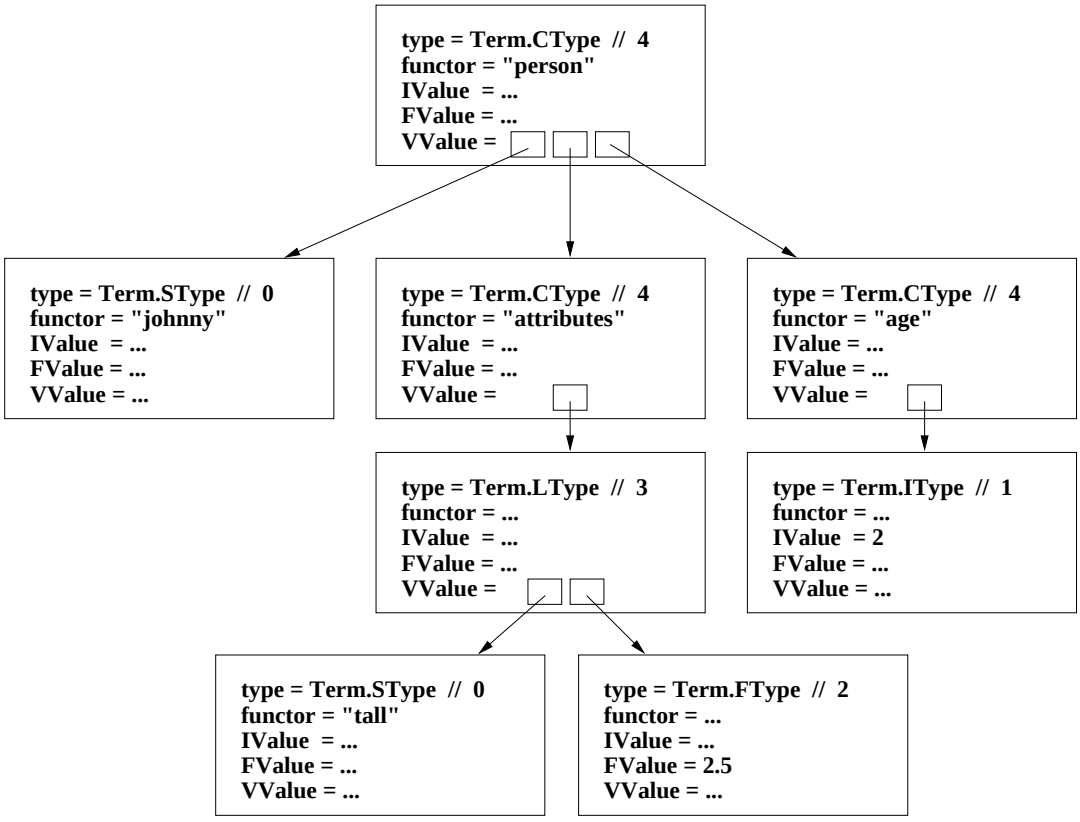

Figure B.1: Underlying representation for a specific term

Figure B.1 shows the internal representation of the following term:

```
person(johnny, attributes([tall, 2.5]), age(2))
```

This term can be generated using the following compact, high-level Term operation:

```
Term person =
Term.parse("person(johnny,attributes([tall,2.5]),age(2))");
```

However, using low-level Term operation, the same term can be obtain more efficiently, but also more verbosely, using the following code:

```
Term person = new Term("person", Term.CType);
Term attributes = new Term("attributes", Term.CType);
Term list = new Term("", Term.LType);
Term age = new Term("age", Term.CType);

Term johnny = new Term("johnny");
Term tall = new Term("tall");
Term height = new Term(2.5);
Term years = new Term(2);

age.addST(years);
list.addSt(tall).addST(height);
attributes.addST(list);
person.addST(johnny).addST(attributes).addST(age);
```

Appendix C

Java Laws: Working with Message Objects

`Message` objects are placeholders for exchanged messages in Java Laws. The `Message` object stores various payloads: `String`, `byte` array (`byte[]`), and serializable objects (`Object`). Also, a Message object stores the source of a message, its destination, law, and other information.

This object has been exposed mainly to provide a compact alternative to writing Java laws using overloaded event methods with different data types. First, laws that are only concerned with certain aspects of the communication (e.g. only the parties involved in the communication but not the data itself) can use this object, both in the event method description as well as in the primitive operations employed. Second, this object allows a brief treatment of the exception event especially for the purpose of recording the failure causes.

A Message object is accessed by reading/writing its fields directly, without setter/getter or specific methods. The following fields are directly accessible in a Message object:

- `int type`: this field specifies the type of message: sent or sent-export(Const.SND or Const.SNDE), forward (Const.FWD), exception (Const.EXC), submitted (Const.SBMT, Const.SBMTC) etc. Depending on the type of message in question, only some of the following fields should be initialized. The rest of the fields are either ignored or not initialized.

- `int p_type`: this field specifies the type of payload this message carries: String (Const .SPLD), byte array (Const.BPLD), or Object (Const.OPLD). Depending on this field exactly one of the s_payload, b_payload, o_payload fields are valid.

- `String s_payload`: carries the String payload of a message when p_type is Const.-SPLD.

- `byte[] b_payload`: carries the byte array payload of a message when p_type is Const.BPLD.
- `Object o_payload`: carries the Object payload of a message when p_type is Const.-OPLD.
- `String source`: carries the source of the message.
- `String dest`: carries the destination of the message.
- `String s_lname`: carries the name of the law the source of a message operates under.
- `String s_hash`: maintains the hash of the source law.
- `String d_hash`: maintains the hash of the destination law.
- `int sport` Holds the destination port number for submitted/release messages
- `String fcause`: carries the failure cause in the case of exception messages.

Appendix D

Remote Synchronous Communication and Java Laws: Working with MethodCall Objects

`MethodCall` objects represent placeholders for method calls in RRMI. MethodCall objects are exposed in the regulated event methods of the RRMI laws, and allow for the retrieval and modifications of the parameters of the in-transit calls, as well as the initiation of new method calls by the controller.

MethodCall offers primarily a friendly format for retrieving the name of the methods and its signature, as well as the signature of the return type of a methods. It also allows the access to a low-level object, `MethodObj` that maintains the value of the arguments or the return value/exception associated with a given call.

The MethodCall class has the following public member fields; they can be accessed directly, or through a number of convenient methods:

- `String name`: represents the name of the method.
- `String[] argtypes` represents the types of the arguments of the method, in the order they appear in the argument list.
- `String retype` represents the data type of the return value or `void` if none.
- `MethodObj mo` represents the underlying object carrying the data (call arguments or return value) during a remote call.

The following represent the methods provided by the MethodCall class:

- `MethodCall(MethodObj mo)`: this constructor creates a new MethodCall object by copying the name, argument types, return type into the fields of the MethodCall, as well as setting the mo reference to the MethodObj argument.

- `getSig()`: returns the signature of the method call.
- `setSig(String sig)` sets the signature of this method call to the given argument
- `getName()`: returns the name of the method stored in the name field.
- `setName(String name)`: sets the name filed with the value of the argument.
- `getArgType(int i)`: returns the type of the argument number i from the argtypes field.
- `setArgType(String type, int i)`: sets the type of the argument number i in the argtypes field.
- `getArg(int i)`: returns the value of the argument number i stored in the MethodObj mo member object.
- `setArg(Object arg, int i)`: sets the value of the argument number i stored in the MethodObj mo member object.
- `setArgs(Object[] args)` sets the values of all the arguments carried by the MethodObj mo member object.
- `getResType()`: returns the type of the result from the retype field.
- `setResType(String type)`: sets the type of the result in the retype field.
- `getResult()`: returns the value of the result of a method call as stored in the MethodObj mo member object.
- `setResult(Object ret)`: sets the value of the result of the method call in the MethodObj mo member object. Note that setting the value of the result and the type of the result to incompatible types, will result in an exception in the stub/skeleton in the application.
- `getException()`: returns the value of the exception result of a method call as stored in the MethodObj mo member object.

- `setException(Exception ex)`: sets the value of the exception result of the method call in the MethodObj mo member object.
- `isCall()`: returns true if this MethodCall is on the calling path, or false if it is on the returning path of a remote call.
- `isReturn()`: returns false if this MethodCall is on the calling path, or true if it is on the returning path of a remote call.
- `isException()`: returns true if this MethodCall object represents a returning call and if the returning call represents an exception; returns false otherwise.
- `setReturn()`: modifies this MethodCall object in order to represent a returning call instead of a calling path. Note that this method might leave this object in an inconsistent state if the result value itself is not set as well.
- `setCall()`: modifies this MethodCall object in order to represent a calling path instead of a returning call. Note that this method might leave this object in an inconsistent state if the argument values are not set as well.
- `getExtra()`: returns the value of the extra argument carried out in the MethodObj mo member object. An extra argument might be carried out along a method call in order to provide support for piggybacking information between controllers.
- `setExtra(Object extra)`: sets the value of the extra argument stored in MethodObj mo member object.

The `MethodObj` object contains the actual data propagated along a remote method call in RRMI. For simplicity, the object represents a bare-bone data structure with no methods, whose member fields can be accessed and modified directly. The following are the member fields of the object:

- `String msig`: represents the signature of the method, containing the result type as well.
- `Object[] args`: represents an array of values for the arguments of the call during a calling path; by default it is null on the return path.
- `int objid`: the object id of the server object.
- `int callid`: the call id of the current call.
- `Object retval`: the value of the return of the method call during the returning path; by default it is null on the calling path.
- `boolean fb`: flag marking the path of the call: true on the calling path and false on the return path.
- `boolean nr`: flag indicating a normal return of a call: true if the method indicates a normal result, or false in the case of an exception.
- `Exception e`: the value of the exception in the case of an exceptional termination of a call; by default null on the calling path and on the return path with a normal result.
- `Object extra`: an extra object carried out along a call for the purpose of piggybacking information between controllers for the purpose of coordinating the access control decision; ignored at the stub and skeleton.

References

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

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

[3] J. R. Anderson, "A security policy model for clinical information systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[4] X. Ao and N. Minsky, "On the role of roles: from role-based to role-sensitive access control," in *Proc. of the 9th ACM Symposium on Access Control Models and Technologies, Yorktown Hights, NY, USA*, June 2004. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[5] X. Ao, N. Minsky, and V. Ungureanu, "Formal treatment of certificate revocation under communal access control," in *Proc. of the 2001 IEEE Symposium on Security and Privacy, May 2001, Oakland California*, May 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[6] X. Ao and N. H. Minsky, "Flexible regulation of distributed coalitions," in *LNCS 2808: the Proc. of the European Symposium on Research in Computer Security (ESORICS) 2003*, October 2003. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[7] X. Ao and N. H. Minsky, "Regulated delegation in distributed systems," in *POLICY '06: Proceedings of the Seventh IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'06)*, (Washington, DC, USA), IEEE Computer Society, 2006, pp. 215–226.

[8] X. Ao, N. H. Minsky, and T. Nguyen, "A hierarchical policy specification language, and enforcement mechanism, for governing digital enterprises," in *Proc. of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks Monterey California*, June 2002. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[9] AspectJ Team, "AspectJ 1.5 Programming Guide," in *http://www.eclipse.org/aspectj/doc/released/progguide/ index.html*, 2006.

[10] O. Bandmann, M. Dam, and B. Firozabadi, "Constrained delegations," 2002.

[11] S. M. Bellovin, "Distributed firewalls," *;login:*, vol. 24, no. Security, November 1999.

[12] K. Beznosov and Y. Deng, "A framework for implementing role-based access control using CORBA security service," in *ACM Workshop on Role-Based Access Control*, 1999, pp. 19–30.

[13] A. Birrell and J. B. Nelson, "Implementing remote procedure calls," *ACMTOCS*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[14] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The keynote trust-management systems, version 2. ietf rfc 2704," Sep 1999.

[15] D. Brewer and M. Nash, "The Chinese Wall security policy," in *Proceedings of the IEEE Symposium in Security and Privacy*, IEEE Computer Society, 1989.

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

[17] Y. Chen, C. Serban, W. Zhang, and N. H. Minsky, "Towards a decentralized and secure electronic marketplace," in *Proc. of the IADIS International Conference on E-Commerce (IADIS'05), Porto, Portugal*, December 2005.

[18] Y. Chen, C. Serban, W. Zhang, and N. H. Minsky, "Towards decentralized and secure electronic marketplace," in *Proc. of the DIMACS Workshop on Security of Web Services and E-Commerce Rutgers University, NJ, USA*, May 2005. (to be presented).

[19] D. Clark and D. Wilson, "A comparison of commercial and military computer security policies," in *Proceedings of the IEEE Symposium in Security and Privacy*, IEEE Computer Society, 1987, pp. 184–194.

[20] W. Clocksin and C. Mellish, *Programming in Prolog*. Springer-Verlag, 1981.

[21] F. Curbera, B. J. Krämer, and M. P. Papazoglou, editors, *Service Oriented Computing (SOC), 15.-18. November 2005*, volume 05462 of *Dagstuhl Seminar Proceedings*, The International Conference and Research Center for Computer Science(IBFI), Schloss Dagstuhl, Germany, 2006.

[22] C. N. da Cruz Ribeiro and P. Ferreira, "A policy-oriented language for expressing security specifications," *International Journal of Network Security*, vol. 5, no. 3, pp. 299–316, 2007.

[23] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in M. Sloman, editor, *Proc. of Policy Worshop, 2001, Bristol UK*, January 2001.

[24] N. Dulay, N. Damianou, E. Lupu, and M. Sloman, "A policy language for the management of distributed agents," in *AOSE*, 2001, pp. 84–100.

[25] C. Ellison, "The nature of a usable pki," *Computer Networks*, no. 31, pp. 823–830, November 1999.

[26] D. Ferraiolo, J. Barkley, and R. Kuhn, "A role based access control model and reference implementation within a corporate intranets," *ACM Transactions on Information and System Security*, vol. 2, no. 1, February 1999.

[27] S. Foley, "The specification and implementation of 'commercial' security requirements including dynamic segregation of duties," in *Proceedings of the 4th ACM Conference on Computer and Communications Security*, April 1997.

[28] I. Foster, C. Kesselman, and S. Tuecke, "The Nexus task-parallel runtime system," in *Proc. 1st Intl Workshop on Parallel Processing*, pp. 457–462, Tata McGraw Hill, 1994.

[29] S. Godic and T. Moses, "Oasis extensible access control markup language (xacml), vesion 2.," March 2005. http://www.oasis-open.org/committees/xacml/index.shtml.

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

[31] J. Gosling, B. Joy, G. Steele, and G. Bracha, “The java language specification, third edition,” *http://java.sun.com/docs/books/jls*, June 2005.

[32] R. Hayton, J. Bacon, and K. Moody, “Access control in an open distributed enviroment,” in *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.

[33] J. A. Hine, W. Yao, J. Bacon, and K. Moody, “An architecture for distributed oasis services,” in *Proceedings IFIP/ACM International Conference on distributed systems platforms*, 2000, pp. 104–120.

[34] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subramanian, “Flexible support for multiple access control policies,” *ACM Trans. on Database Systems*, vol. 26, no. 2, pp. 214–260, June 2001.

[35] J.-J. Jeng, H. Chang, and J.-Y. Chung, “A policy framework for web-service based business activity management (bam),” *Inf. Syst. E-Business Management*, vol. 2, no. 1, pp. 59–88, 2004.

[36] J. B. D. Joshi, E. Bertino, B. Sahfiq, and A. Ghafoor, “Dependencies and separation of duty constraints in gtrbac,” in *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, 2003.

[37] M. Juric, I. Rozman, M. Hericko, and T. Domajnko, “Corba, rmi and rmi-iiop performance analysis and optimization,” in *SCI 2000, Orlando, Florida, USA*, July 2000.

[38] G. Karjoth, “The authorization service of tivoli policy director,” in *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *European Conference on Object-Oriented Programming vol.1241, pp.220-242*, 1997.

[40] M. Little and S. Shrivastava, “An examination of the transition of the arjuna distributed transaction processing software from research to products,” in *Proceedings of the 2nd USENIX Workshop on Industrial Experiences with Systems Software (WIESS ’02), Boston, MA, USA, 8 December 2002 (Co-located with OSDI ’02) USENIX Association 2002*, 2002.

[41] Microsoft Corporation, “Com: Component object model technologies,” http://www.microsoft.com/com/default.mspx.

[42] N. Minsky, “Law-governed systems,” Technical Report LCSR-TR-101, Department of Computer Science, Rutgers University, Feb. 1987.

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

[44] N. Minsky, “Law-governed regularities in object systems; part 1: An abstract model,” *Theory and Practice of Object Systems (TAPOS)*, vol. 2, no. 1, 1996.

[45] N. Minsky, “Law governed interaction (lgi): A distributed coordination and control mechanism (an introduction, and a reference manual),” Technical report, Rutgers University, June 2005. (available at `http://www.moses.rutgers.edu/documentation/manual.pdf`).

[46] N. Minsky and P. Pal, "Law-governed regularities in object systems; part 2: A concrete implementation," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 2, 1997.

[47] N. Minsky and V. Ungureanu, "Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems," *TOSEM, ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 3, pp. 273–305, July 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

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

[49] M. Moyer and M. Abamad, "Generalized role-based access control," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001, pp. 391–398.

[50] Ninja Team, "The ninja project enabling internet-scale services from arbitrarily small devices," http://ninja.cs.berkeley.edu/.

[51] OASIS Service Oriented Architecture Technical Committee, "Oasis reference model for service oriented architecture v 1.0," 2006. (available at `http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf`).

[52] Object Management Group, "Omg security," *http://www.omg.org/ technology/documents/ formal/omg _security.htm*.

[53] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer, "Service-oriented computing: A research roadmap," in *Service Oriented Computing*, 2005.

[54] I. Ray, "Applying semantic knowledge to real-time update of access control policies," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 844–858, 2005.

[55] I. Ray and T. Xin, "Concurrent and real-time update of access control policies.," in *14th International Conference of Database and Expert Systems*, 2003, pp. 330–339.

[56] T. Ryutov and C. Neuman, "Representation and evaluation of security policies for distributed system services," in *In Proceedings of the DARPA Information Survivability Conference and Exposition, South Carolina*, January 2000, pp. 172–183.

[57] R. Sandhu, V. Bhamidipati, and M. Munawer, "The ARBAC97 model for role-based administartion of roles," *ACM Transactions on Information and System Security*, vol. 2, no. 1, pp. 105–135, Feb. 1999.

[58] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The nist model for role-based access control: Towards a unified standard," in *Proceedings of ACM Workshop on Role-Based Access Control*, ACM, July 2000.

[59] R. S. Sandhu, "A lattice interpretation of the chinese wall policy," in *Proc. 15th NIST-NCSC National Computer Security Conference*, 1992, pp. 329–339.

[60] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi, "OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP," *Lecture Notes in Computer Science*, vol. 2104, pp. 130–??, 2001.

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

[62] C. Serban, “The law specification and api,” Technical report, Rutgers University, November 2005. (available at `http://www.moses.rutgers.edu/api`).

[63] C. Serban, “The lgi web-site,” Technical report, Rutgers University, June 2005. (available at `http://www.moses.rutgers.edu`).

[64] C. Serban, “The implementation and the deployment of the pay-per-service rrmi law,” Technical report, Rutgers University, November 2006. (available at `http://www.moses.rutgers.edu/rrmi/examples/payperservice/`).

[65] C. Serban, X. Ao, and N. Minsky, “Establishing enterprise communities,” in *Proc. of the 5th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2001), Seattle, Washington*, September 2001. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).

[66] C. Serban, Y. Chen, W. Zhang, and N. H. Minsky, “The concepts of a decentralized and secure electronic marketplace,” *Electronic Commerce Research Journal, ECR, Springer-Verlag*, no. 1, 2008.

[67] C. Serban and N. Minsky, “Regulating synchronous communication, and its applications to web-services,” in *DIMACS Workshop on Security of Web Services and E-Commerce*, 2005.

[68] C. Serban and N. H. Minsky, “Generalized access control of synchronous communication,” in *Middleware*, 2006, pp. 281–300.

[69] C. Serban and S. Tyszberowicz, “Enforcing interaction properties in aosd-enabled systems,” in *ICSEA ’06: Proceedings of the International Conference on Software Engineering Advances*, (Washington, DC, USA), IEEE Computer Society, 2006, p. 8.

[70] C. Serban and S. Tyszberowicz, “Aspectjtamer: The controlled weaving of independently developed aspects,” in *Proc. of the IEEE International Conference on Software-Science, Technology, and Engineering (SwSTE’07), Herzliya, Israel*, October 2007.

[71] Sun Microsystems, “Java Remote Method Invocation (RMI),” http://java.sun.com/products/jdk/rmi/index.html.

[72] Sun Microsystems, “RMI Wire Protocol,” *http://java.sun.com/ j2se/1.4.2/docs/ guide/rmi/ spec/rmi-protocol.html*.

[73] W. Wulf, E. Cohen, W. Corwin, A. Jones, C. Levin, C. Pierson, and F. Pollack, “Hydra: The kernel of a multiprocessor operating system,” *Communications of the ACM*, vol. 17, pp. 337–345, 1974.

[74] W. Zhang, C. Serban, and N. Minsky, “Establishing global properties of multiagent systems via local laws,” 2007.

Vita

Constantin Serban

Education

Ph.D. Computer Science, Rutgers University, New Jersey 2008

M.S. Computer Science and Systems, Polytechnic University Bucharest, Romania 1997

B.S. Computer Science and Systems, Polytechnic University Bucharest, Romania 1996

Publications

- Constantin Serban, Yingying Chen, Wenxuan Zhang, and Naftaly H. Minsky. "The Concepts of a Decentralized and Secure Electronic Marketplace". In the Electronic Commerce Research Journal, ECR, Springer-Verlag , 2008.
- Constantin Serban, and Shmuel Tyszberowicz. "AspectJTamer: The Controlled Weaving of Independently Developed Aspects". In the Proceedings of the IEEE International Conference on Software — Science, Technology and Engineering, SwSTE07.
- Wenxuan Zhang, Constantin Serban, and Naftaly Minsky. "Establishing Global Properties of Multiagent Systems via Local Laws", In the "Environments for the Multiagent Systems III, Van Parunak, Fabien Michel, and Danny Weyns (Editors)", LNCS series, Springer-Verlag, 2007.
- Constantin Serban, and Naftaly H. Minsky. "Generalized Access Control of Synchronous Communication". In the Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference, Middleware 2006.
- Constantin Serban, and Shmuel Tyszberowicz. "Enforcing Interaction Properties in AOSD-Enabled Systems". In the Proceedings of the IEEE International Conference of Software Engineering Advances, ICSEA 2006.
- Yingying Chen, Constantin Serban, Wenxuan Zhang, and Naftaly H. Minsky. "Towards a Decentralized and Secure Electronic Marketplace". In the Proceedings of the IADIS International E-Commerce Conference, IADIS 2005.
- Constantin Serban, Xuhui Ao, Naftaly H. Minsky. "Establishing Enterprise Communities". In the Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2001.