Generalized Access Control of Synchronous Communication[⋆]

Constantin Serban and Naftaly Minsky

Rutgers University, Piscataway, NJ 08854 USA,
{serban,minsky}@cs.rutgers.edu

Abstract. The security of modern networked applications, such as medical institutions or commercial enterprises, requires increasingly sophisticated access control (AC) that can support *communal* (e.g., enterprise wide) and *stateful* (i.e., sensitive to the history of interaction) policies. The Law-Governed Interaction (LGI) mechanism supports such policies, but so far only for asynchronous message passing communication. This paper extends LGI to synchronous communication, thus providing sophisticated control over this important and popular mode of communication. Among the novel characteristics of this control are: the regulation of both the request and the reply—separately, but in a coordinated manner; *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.

Keywords: Access-control, Security, RMI, Synchronous communication, Law Governed Interaction
Technical area(s): Security

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 must communicate via the Internet, and can no longer protect themselves by hiding within their local intranet, behind their firewalls. Rather, they now depend on access control to protect them against malicious attacks, by regulating the messages exchanged among their people and components and between them and the outside. Simultaneously, the complexity of access control is increasing due to the following needs: (a) the need to support increasingly sophisticated policies; (b) the need to regulate the interactions among the members of large and distributed communities of agents, via *communal* (overarching) policies; (c) the need to provide for *interoperability* between policies; and (d) the need to organize policies into *hierarchies*, in order to regulate complex

[⋆] Work supported in part by NSF grant No. CCR-04-10485 systems such as those that serve enterprises—with their intrinsically hierarchical governance—and *federations* of enterprises, as in *grid computing*.

In previous papers [18,2,3] we have shown how these needs can be addressed by Law-Governed Interaction (LGI), which 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. LGI is a significant generalization of the conventional concept of access-control. It also represents a radical departure from conventional AC mechanism in that it employs an inherently *decentralized* policy-enforcement technique.

However, LGI has been defined 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]. We will argue in this paper that the control of synchronous communication requires different treatment than that of asynchronous one, particularly when dealing with communal and stateful policies. This is because there are some special needs that arise when regulating synchronous communication, which include: (a) the need to control both the request and the reply parts of a call, separately, but in a coordinated fashion; and (b) the need to provide clients with a *regulated timeout* capability, taking into account the concerns of both the server and the client. This paper addresses these needs, by extending LGI to support synchronous communication.

The rest of this paper is organized as follows. Section 2 motivates this paper by explaining some of the needs to generalize access control, particularly for synchronous communication—illustrating them via a case study, which will be used throughout the paper. Section 3 provides a brief overview of the concept of LGI. In Section 4 we describe the architecture of the proposed AC mechanism for synchronous communication, and we show how it supports the policy introduced in the case study. Section 5 describes the implementation of this mechanism for the RMI protocol, giving rise to what we call "Regulated RMI" (or RRMI). Section 6 discusses related work, and Section 7 concludes this paper.

2 On the Need for a Generalized Access Control Mechanism

We elaborate here on several needs of modern computing for a generalized AC mechanism. We start with a brief discussion of the need for more expressive poli-

[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; the latter has been chosen as a starting point for the proof of concept implementation in this paper.

cies, for communal policies, and the need for decentralization; these properties apply to both asynchronous and synchronous communication equally, and they are already supported for message-passing communication in the previous version of LGI. We then discuss in greater detail the special needs of synchronous communication, namely the need to control both the request and the reply parts of a call, and the need to provide clients with a *regulated timeout* capability. We will motivate and start these discussions with a simple case study.

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

In order to illustrate the types of policies we have in mind for the regulation of synchronous communication, 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*. This means that 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.

2.2 The Need for more Expressive Policies

While the conventional access control mechanisms are still largely based on the *access control matrix* model, often upgraded into "role-based AC" (RBAC) [21], the limitations of this model have been long recognized in the context of commercial [7] and clinical [1] applications. These limitations are also becoming increasingly apparent in other application domains. We point out here two important, and closely related, features that are missing in the traditional AC model.

One such feature is *sensitivity to the history of interaction*, which gives rise to the so called *stateful*, or *dynamic* policies. Our example policy PPS is stateful in this sense, as 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 made. Such budgetary controls are, of course, critical in financial systems, but they are important in other kinds of systems as well. Other types of stateful policies include, in particular, *dynamic separation of duties* [10], and the so called Chinese-Wall policy.

The state representing the history of interaction which is relevant to the policy at hand is called *control state*. The control state may include for example such things as the e-wallets of the agents subject to policy PPS.

Another important feature, missing from traditional AC, is a degree of *initiative* that a policy can take. Conventional AC policies are limited to permitting or prohibiting messages. But one often needs to associate other actions with the sending or receipt of a message, such as sending a copy of the message to some audit trail server, or changing the state of the sender or receiver if the policy is stateful, as is required by our PPS policy above. Some of these capabilities has been introduced into several recent AC models. In particular, the AC model of Ryutov and Neuman [20] supports policies that can exhibit simple initiatives, but they do not support stateful policies; the same is true for XACML [12], a recent AC standard for web-services.

2.3 The Need for Communal Policies

Most conventional AC mechanisms are designed for *server-centric* policies. Such a policy is employed by an individual server to regulate the use of own its resources by its clients. Such a policy is usually expressed via Access Control Lists, or via a formalism like Keynote [6]. 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 their clients—or a community of peers—is subject to an overarching *communal* policy. Our example 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 AC domain, such as the enterprise.

2.4 The Need for Decentralization

The importance of communal, enterprise-wide policies has been recently recognized by some academic projects [9], as well as by commercial systems such as IBM-Tivoli [14], and by XACML [12]. They all 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 [2], 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.

We believe, therefore, that for an AC mechanisms to support communal and stateful policies in a scalable manner, it needs to be *decentralized*. As we shall see in Section 3, such decentralization can be accomplished efficiently and scalably by associating with every agent x a private reference monitor, called *controller*, that mediate all the interaction of x with other members of the community governed by the policy at hand. This controller also maintains the *local control state* of agent x, which reflect the history of its interaction with the rest of the community in question. It is this control state which would maintain the e-wallet of x under the PPS policy above.

2.5 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 kinds 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.

An example of such sensitivity of a policy to the reply is provided by the PPS policy introduced in Section 2. 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 of Section 2, 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.

2.6 The Need to Regulate Timeouts

Under synchronous communication the client thread is blocked until it gets the reply. This feature is intended to provide transparency of the network communication, by making remote calls appear to programmers as local calls [5]. 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—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, because it has been notified *a priori* of the time limit.

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 indicate to the client 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.

A policy regarding unplanned timeouts is just what is provided for by Point 3 of the PPS policy in Section 2. 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 4.1.

3 An Overview of LGI

Broadly speaking, 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. By "actor" we mean an arbitrary process, whose structure and behavior is left unspecified. An actor engaged in an LGI-regulated interaction, under a law $\mathcal{L}$, is called an $\mathcal{L}$-*agent* (or simply an "agent," when the identity of the law does not matter); 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 an $\mathcal{L}$-community.

LGI thus turns a set of disparate actors, which may not know or trust each other, into a *community* of agents that can rely on each other to comply with the given law $\mathcal{L}$. This is done via a distributed collection of generic components called *private controllers*, one per $\mathcal{L}$-agent, which need to be *trusted* to mediate all interactions between these agents, subject to a specified law $\mathcal{L}$ (as illustrated in Figure 1).

The private controllers are actually hosted by what we call *controller pools*—each of which is a process of computation that can operate several (in the hundreds) private controllers, thus serving several different agents, possibly subject to different laws[2]. A prototype of LGI was released in October 2005 [17]; this section provides only a very brief overview of LGI. For more information, the

[2] We often use the term "controller" for either a controller-pool or for a private-controller—expecting the ambiguity to be resolved by the context.

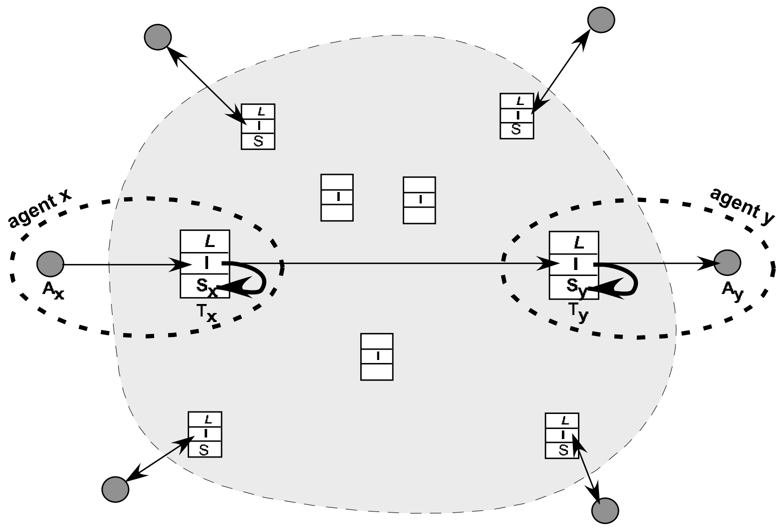


Fig. 1. Interaction via LGI: Actors are depicted by circles, interacting across the Internet (lightly shaded cloud) via their private controllers (boxes) operating under law L. Agents are depicted by dashed ovals that enclose (actor, controller) pairs. Thin arrows represent messages, and thick arrows represent modification of state.

reader is referred to the LGI tutorial and manual, available through the above mentioned website, and to a host of published papers.

Agents and their Private Controllers: An $\mathcal{L}$-agent x is a pair $x = \langle A_x, T_x^{\mathcal{L}} \rangle$, where A_x is an *actor*, and $T_x^{\mathcal{L}}$ is its *private controller*, which mediates the interactions of A_x with other LGI-agents, subject to law $\mathcal{L}$. The role of the controllers is illustrated in Figure 1, which shows the passage of a message from an actor A_x to A_y, as it is mediated by a pair of controllers, first by $T_x^{\mathcal{L}}$, and then by $T_y^{\mathcal{L}}$—both operating, in this case, under the same law.

The Structure and Operations of Private Controllers: Broadly speaking, a private controller, such as $T_x^{\mathcal{L}}$ above, can be described as a triple $\langle I, \mathcal{L}, S_x \rangle$ (depicted by boxes in Figure 1), where I is a generic interpreter and enforcer of LGI laws; $\mathcal{L}$ is the law under which this particular controller operates; and S_x is the *control state* (or, "cState") of agent x, whose content, semantics, and dynamic behavior are largely defined by law $\mathcal{L}$. The concept of law is defined in the following section.

To describe the behavior of a controller, we need to introduce its main features. First, a private controller $T_x^{\mathcal{L}}$ operates by responding to certain *regulated events* that occur at it, which includes, among others: (a) the arrival of various messages at the controller—messages sent by its own actor A_x to other agents, and messages sent by others to agent x; and (b) the coming due of an obligation. Second, a private controller features a set of *primitive operations* that are carried out only if mandated by the law. They include operations on the cState S_x of the agent in question, and operations that cause messages to be forwarded to other agents.

A controller $T_x^{\mathcal{L}}$ operates *sequentially*, by reacting to the regulated events that occur at it, in the order of their occurrence (and in an arbitrary order for events that occur simultaneously). It reacts to each such event as follows: (a) it evaluates the *ruling* of law $\mathcal{L}$ for this event, where the ruling is a list of primitive operations; and (b) it carries out this ruling, by executing all the operations in it, in the order of their appearance in the ruling, and *atomically*—before the controller turns to the next event.

The Concept of Law, and the Semantics of LGI: Our concept of law differs structurally from the conventional concept of an AC policy (such as that of XACML) mostly in that it is **local**—in the sense that an LGI law can be complied with, by each member of the community subject to it, without having any direct information of the coincidental state of other members. This locality is important because it enables the decentralization of law enforcement, and thus provides for scalability even in the case of stateful policies.

It is important to note that, despite the fact that locality constitutes a strict constraint on the structure of LGI laws, it does not reduce their expressive power, as has been proved in [17]. In particular, despite its *structural locality*, an LGI law can have *global effect* over the entire $\mathcal{L}$-community—simply because all members of that community are subject to the same law—and can, thus, be used to establish *mandatory*, community-wide constraints.

The following is an **abstract definition** of LGI laws:

> *A law $\mathcal{L}$ is 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 any possible control-state s.*

Note that the ruling of the law is not limited to accepting or rejecting a message, but can mandate any number of operations, providing laws with a strong degree of *initiative*, as discussed in the introduction. Also, the operations that can be included in the ruling may update the cState of the agent, thus providing for *stateful policies.* Finally, the ruling may impose an *obligation* on the agent, which provides a *proactive capability.*

This definition is abstract in that it is independent of the language used for specifying laws. We currently use two such languages—one is based on Prolog, and the other one on Java. But despite the pragmatic importance of a particular language being used for specifying laws, the semantics of LGI is basically independent of that language.

On the Basis for Trust Between Members of a Community: In order for an agent x to trust its peer y to operate under the same law $\mathcal{L}$, it is sufficient to have the assurance that the following three conditions are satisfied: (a) the exchange between x and y is mediated by *bona fide* private controllers T_x and T_y, respectively; (b) both controllers operate under law $\mathcal{L}$; and (c) the $\mathcal{L}$-messages exchanged between x and y are transmitted securely over the network.

The first of these conditions is the hardest to satisfy. LGI ensures this condition via certification. That is, a given law may require the controllers interpreting it to authenticate themselves by means of a certificate signed by a specified certification authority (CA). Such a CA that is willing to certify the controllers as correct is presumably associated with some reputed organization that manages and maintains an entire set of controllers.

To ensure condition (b), that is that the interacting controllers T_x and T_y operate under the same law, LGI adopts the following protocol: a controller T_x appends an *one way hash* [19] `H` of its law to every message it controls. The controller of the receiving peer, T_y, would accept this as a valid $\mathcal{L}$-message only if `H` is identical to the hash of its own law. Of course, such an exchange of hashes can be trusted only if condition (a) is satisfied.

Finally, to ensure the validity of condition (c) above, the messages sent across the internet—between actors and their controllers, and between pairs of controllers—should be digitally signed and encrypted. These conventional but rather expensive measures can be dispensed with if one is not concerned about monitoring and spoofing of messages.

4 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 2. This is similar to the manner that conventional AC mechanisms for asynchronous communication operate.

In this paper 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. 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 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 this AC mechanism is discussed in Section 5. In this section we will show how this mechanism can be used to implement the PPS policy of Section 2.1. For this purpose, we will express a law that implements this policy via a pseudocode; the formalization in the Java-based law language of

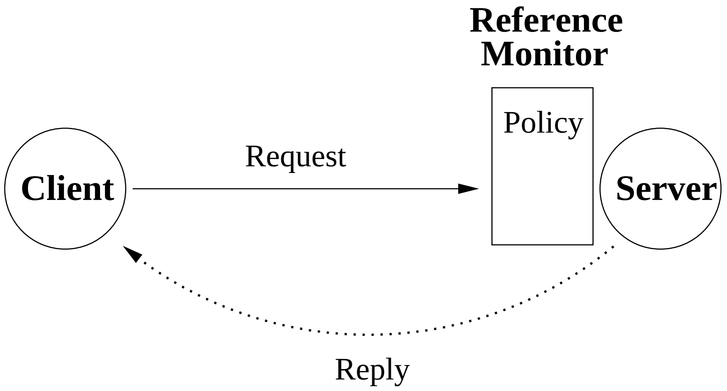


Fig. 2. Server Centric Access Control over RPC

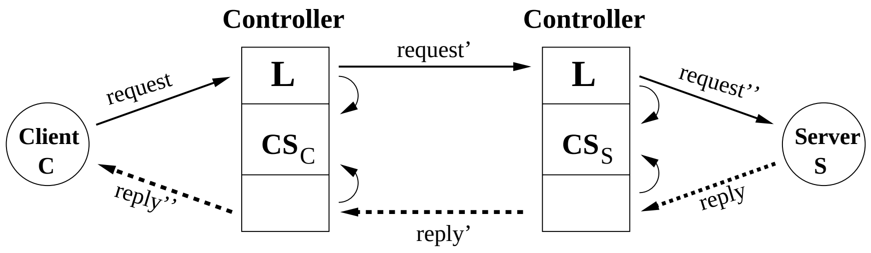


Fig. 3. Regulated Synchronous Communication

LGI, defined in [17] is available at: `http://www.moses.rutgers.edu/rrmi/examples/payperservice/` . This pseudocode consist of rules of the following type:

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

Each of these rules has three parts, briefly described below.

The `event` part of a rule specifies one of the events that may occur at a controller. Below are the four events which are directly involved in synchronous communication:

– *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 arrived 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 forwarding of a message or the modification of the control state. The modification of the control state is critical because 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.

4.1 The Implementation of the *PPS* Policy

Access to services under our Pay-Per-Service (PPS) policy introduced informally in Section 2.1 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.

Figure 4 presents 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 `arrivedCall(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 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

```
R1  upon sentCall(getBudget)       :   do forwardCall
R2  upon arrivedCall( getBudget)   : if role == budgetOfficer
                                       do forwardCall
                                     else
                                       do forwardResult(Exception(NotBOfficer))
R3  upon sentResult(getBudget)     :   do forwardResult
R4  upon arrivedResult(getBudget)  :   do addEWallet(method.result)
                                       do forwardResult
R5  upon sentCall( S )             : if eWalletAmnt < serviceCost
                                       do forwardResult(Exception(OutOfCurrency))
                                     else
                                       do removeEWallet(serviceCost)
                                       do addEscrow
                                       do forwardCall
R6  upon arrivedCall(S)            :   do addEscrow
                                       do forwardCall
R7  upon sentResult( S )           : if method.result is Exception
                                       do removeEscrow
                                       do forwardResult
                                     else
                                       do addEWallet(serviceCost)
                                       do removeEscrow
                                       do forwardResult
R8  upon arrivedResult( S )        : if method.result is Exception
                                       do addEWallet(serviceCost)
                                       do removeEscrow
                                       do forwardResult
                                     else
                                       do removeEscrow
                                       do forwardResult
R9  upon sentCall(cancel)          :   do forwardCall
R10 upon arrivedCall( cancel )     : if escrow.exists()
                                       do addEWallet(f(serviceCost))
                                       do removeEscrow
                                       do forwardResult
                                       do forwardResult(Exception(Cancelled))
                                     else
                                       do forwardResult(Exception(NoPendingCall))
R11 upon arrivedResult( cancel )   :   do forwardResult
R12 upon arrivedResult( S )        : if method.result is Exception(Cancelled)
                                       do addEWallet(serviceCost-f(serviceCost))
                                       do removeEscrow
                                       do forwardResult
```

Fig. 4. Pay-per-service Law

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.

5 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). 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 in Java. The use of Java for writing laws and their generic structure is described in detail in [17]. The access control rules are expressed in RMI laws by mapping the events introduced in Section 4 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. An example of a formal RMI law implementing the PPS policy is available at:
`http://www.moses.rutgers.edu/rrmi/examples/payperservice/` .

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 (or target). 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 changed the transport protocol to our version of LGI-controllable transport layer. As opposed to JRMP, this new transport layer provides enough in-transit information permitting an adequate 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 [24]. This is an open source RMI implementation developed as

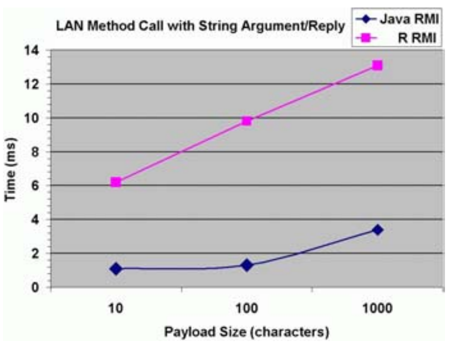
```
              /*remote object code*/
public class RecordServerImpl extends LgiRemoteObject implements RecordServer{
  public RecordServerImpl(PMember principal) throws RemoteException{
    super(principal);
  }
  public String getRecord() {
    ...// specific code
  }
}
              /*exporting 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
```

Fig. 5. Sample RRMI client-server code

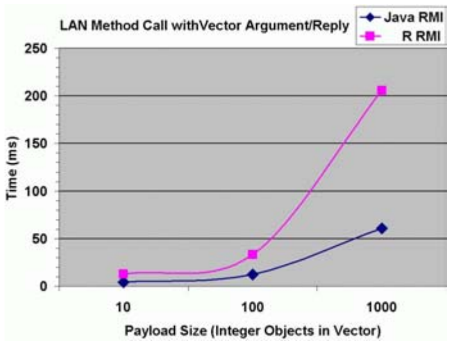
part of the Ninja project at Berkeley, and is source-level compatible with Java RMI.

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

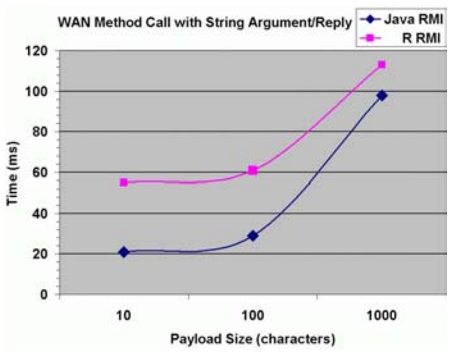
On the Performance of RRMI: We compared the performance of RRMI implementation with standard Java RMI/JRMP. The objective of our performance



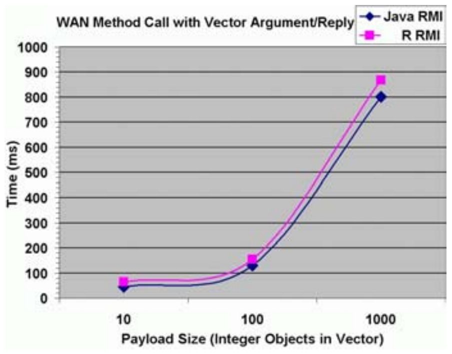
(a) String Transfer-LAN



(b) Vector Transfer-LAN



(c) String Transfer-WAN



(d) Vector Transfer-WAN

Fig. 6. RRMI vs. JavaRMI/JRMP Performance Comparison

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 100Mhz 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 6 (a) and Figure 6 (c) 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 6 (b) and Figure 6 (d) shows 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 marshaling/unmarshalling are, by far, the dominant time-consuming component of an RMI call, and our solution requires the additional marshaling and serialization operations by two controllers. Additionally, Java RMI is optimized for communication of strings and small payloads, while RRMI incurs the constant penalty of carrying extra security-related payloads. As observed in Figure 6 (c) and (d), in the case of SANs or WANs, this disadvantages are offset by the large communication latency. Given the added value of our mechanism, the results are very encouraging. At the same time the results prove our implementation to be comparable or better than RMI/IIOP, as reported in [16], 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.

6 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.

Predefined timeout is not available under Sun RPC, Java RMI [23], and DCOM [8]. These middlewares rely on the underlining network stream timeout (which is neither explicit nor predictable). Under CORBA [13,4], 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 [15] [22] [11]. 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.

7 Conclusions

This paper introduces an extension of LGI which allows sophisticated and scalable regulation of synchronous communication. The following are some of the notable characteristics of the resulting regulation model: (a) it regulates both the request part and the reply part of a call; (b) the regulation is done both at the client and at the server side; and (c) it provides control over how the timeout is handled in a manner that can take into account the concerns of both the client and the server. The proposed model for access control has been implemented for Java RMI, giving rise to a mechanism called Regulated RMI (or, RRMI).

The full power of the proposed mechanism resides in its ability to handle stateful and communal policies. However, we believe that this mechanism is useful for access control even under less sophisticated requirements. RRMI can also be used for the customization of synchronous protocols even when the access control is not necessary.

References

1. J. R. Anderson. A security policy model for clinical information systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.
2. 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`).
3. 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`).
4. Konstantin Beznosov and Yi Deng. A framework for implementing role-based access control using CORBA security service. In *ACM Workshop on Role-Based Access Control*, pages 19–30, 1999.
5. A. Birrell and J. B. Nelson. Implementing remote procedure calls. *ACMTOCS*, 2(1):39–59, February 1984.
6. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The keynote trust-management systems, version 2. ietf rfc 2704. Sep 1999.
7. D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium in Security and Privacy*, pages 184–194. IEEE Computer Society, 1987.
8. Microsoft Corporation. Com: Component object model technologies. http://www.microsoft.com/com/default.mspx.
9. 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*, 2(1), February 1999.
10. 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.
11. I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.
12. 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.
13. Object Management Group. Omg security. http://www.omg.org/technology/documents/formal/omg_security.htm.

14. G. Karjoth. The authorization service of tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001.
15. M.C. Little and S.K. 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.
16. Juric M.B., Rozman I., Hericko M., and Domajnko T. Corba, rmi and rmi-iiop performance analysis and optimization. In *SCI 2000, Orlando, Florida, USA*, July 2000.
17. Naftaly 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`).
18. N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000. (available from `http://www.cs.rutgers.edu/~minsky/pubs.html`).
19. R. Rivest. The MD5 message digest algorithm. Technical report, MIT, April 1992. RFC 1320.
20. 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*, pages 172–183, January 2000.
21. R. Sandhu, V. Bhamidipati, and M. Munawer. The ARBAC97 model for role-based administartion of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, February 1999.
22. Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP. *Lecture Notes in Computer Science*, 2104:130–??, 2001.
23. Inc Sun Microsystems. Rmi wire protocol. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-protocol.html.
24. Ninja Team. The ninja project enabling internet-scale services from arbitrarily small devices. http://ninja.cs.berkeley.edu/.