MAKING SOA-BASED SYSTEMS COHERENT AND TRUSTWORTHY

BY TIN LAM

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

May, 2012

ABSTRACT OF THE DISSERTATION

Making SOA-Based Systems Coherent and Trustworthy

by TIN LAM

Dissertation Director: Naftaly H. Minsky

Under Service Oriented Architecture (SOA), a software system - such as one that supports an enterprise-consists of multiple heterogeneous servers. These servers may be distributed over the internet, and may be managed under different administrative domains. SOA has become hugely popular, particularly as the architecture of large and complex distributed systems such as enterprise systems, grids, virtual enterprises, and supply chains. Unfortunately, this architecture as it is currently defined and being used, suffers from serious problems as outlined below.

First, there is no means to ensure that a fragmented and open system like a SOA-based system satisfies desired global constraints, or can establish any regularity over the system. Second, service providers can make their own commitments to clients. However, the SOA methodology provides no guarantee to clients that the commitment made by a given server will be satisfied. Note that although service providers should have the freedom to specify their own commitments, they must conform to the global constraints imposed at large, and such conformity must be enforced. Third, when using a composite service, clients indirectly communicate with services via an orchestrator. Such indirect interaction may cause concerns for both clients and services, but no formal mechanism has been designed to address those concerns. Finally, it is not possible to ensure that coordination between disparate servers— called ”choreography” under SOA— is carried out safely and correctly.

The overarching goal of my dissertation is to design a regulatory mechanism to address

all these problems in a scalable manner. The mechanism is an extension of Law Governed Interaction (LGI) - a decentralized coordination and control mechanism for distributed systems. We call this mechanism "LG-SOA", for Law-Governed-SOA, which enables high expressive power, efficient enforcement, as well as good scalability. We will also present how LG-SOA can be applied to legacy systems. Case studies in the context of enterprise systems demonstrate the flexibility and applicability of this mechanism. Experiments show the overhead introduced by LG-SOA is relatively small, especially in the context of geographically distributed systems like SOA-based systems. In sum, LGI-SOA is effective and versatile in making SOA-based systems more coherent and trustworthy.

Acknowledgements

I would never have been able to finish my dissertation without the guidance of my committee members, help from friends, and support from my family.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Naftaly Minsky, for his excellent guidance, support and patience. His wisdom, knowledge and commitment to the highest standards inspired and motivated me.

I am very grateful to all the members of my dissertation committee, including Dr. Naftaly Minsky, Dr. Thu Nguyen, Dr. Rebecca Wright, and Dr. Josephine Micallef for their mentorship and guidance.

I would like to thank Constatin Serban, Xuhui Ao and Wenxuan (Bill) Zhang for their implementation of Law-Governed Interaction over the time.

Finally, I would like to thank my family for their support and encouragement. They were always there cheering me up and stood by my through the good times and bad.

Dedication

To My Family

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 The Concept of SOA, and its difficulties

In the modern age, people increasingly depend on advanced computing applications for performing day-to-day activities and for satisfying their communication needs. This brings about the creation of such new applications as enterprise systems, grids, virtual enterprises, and supply chains. Those applications often consist of great numbers of components, heterogeneous in nature and distributed on a large scale across the Internet. Along with the rise of new applications, a new model of software development, called Service Oriented Architecture (SOA), is emerging.

Under SOA, a system is divided into distributed services that are made available to clients (or service consumers) over the Internet. A service is a piece of software that implements some well-defined functionality that can be consumed by clients (e.g., other services). Services communicate with each other by means of message exchanges. Note that services may be designed, constructed, and maintained under different administrative domains. They may be written in different languages and may run on different platforms. Moreover, services may be changed or removed dynamically while new ones may be added into a system an unpredictable manner. The fragmentation and openness of SOA-based systems present serious obstacle to their manageability and reliability.

First of all, because of the heterogeneity, there is a need to impose global constraints on all servers in SOA-based systems. The enforcement of such constraints guarantees uniformity in certain activities of servers. For example, it is necessary to provide system authorities (e.g., system administrators or accountants) with the capabilities to monitor or audit servers' performance and to control certain servers' activities. Equipped with such capabilities, system authorities can steer servers to achieve a system's goal.

Furthermore, individual servers typically make commitments to their clients regarding the services they provide. Such commitments are specified in the server policies, and may include such things as privacy policies, and Service Level Agreements (SLAs) as well as run-time promises to reserve certain merchandise for a specified time period. Yet the SOA methodology provides no guarantees to clients that the commitment made by a given server will actually be satisfied. Note that although service providers should have the freedom to specify commitments to clients in their policies, their policies must conform to the global constraints imposed at large, and such conformity must be enforced.

Under orchestration scheme, an orchestrator can orchestrate existing services to create composite services. In other words, a client which uses those composite ones indirectly interact with orchestrated services, but the client and those servers may not be aware of each other. Such indirect interaction may cause concerns for both clients and the orchestrated servers. On the clients' side, for example, they are afraid that the orchestrator may to abuse the privilege to act on their behalf, or that the orchestrator does not complete certain important things for them, etc. On the side of orchestrated servers, they may choose to consider an orchestrator as a special client, which brings about benefits for all of them. Therefore, the orchestrated services want the orchestrator to fulfill its commitments such as fair treatment among equivalent services, or sincere cooperation in handling client request, etc.

Under choreography scheme, services interact for a common goal. They share responsibility, and none takes the lead as an orchestrator during the interaction. The problem is that leading proposals for choreography modelling is too obtrusive, focusing on the procedural aspects of the collaboration in choreography. We believe choreography model should focus on specifying constraints to make all services interact correctly, allowing them to operate freely as long as they satisfy the choreography constraints. One big challenge is that a service may participate in multiple choreographies after other policies (e.g., global constraints and server policies) are deployed. Moreover, there is no mechanism to enforce choreography descriptions as they are currently modelled. Finally, since a service is not able to determine whether its collaborating services comply with all choreography constraints, effective collaboration cannot be achieved.

1.2 The conventional approach for addressing these difficulties

As discussed in previous section, a typical enterprise is governed by multiple policies which can be defined by different law makers. For example, global constraints, server commitments, choreography constraints are defined by system managers, service providers, and choreography modelers, respectively. Note that there is a conformance relationship between those policies (e.g, policies defined by service provider must conforms to global constraint). Therefore, we need a mechanism that helps (1) organize and classify a set of enterprise policies and (2) describe the conformance of a policy to another. Moreover, in order to capture the semantics and the details of various interactions, stateful policies–policies that are sensitive to the history of interaction– are needed.

But specification of the policy that is to govern a given community is only the first step toward its implementation–the second, and also more critical step is to ensure that all members of the community actually conform to the specified policy. While centralized coordination mechanisms (CCM) can be employed to enforce stateful policies, it is not easily scalable to support large systems. The conventional approach to the implementation of a policy is to build it into all members of the community subject to it. To do this would require good software engineering, human governance, etc. Even though all of the above is useful and necessary, it is not sufficient. If the community in question is large and heterogeneous, then such ”manual” implementation of the policy would be too laborious and error-prone to be practical. That happens because system managers-who are responsible for implementing policies-have little, if any, sway over server programs, which may be dispersed all over the Internet. Moreover, it is also possible that codes of server programs may not be available to the managers for privacy reason; and if they are available, the managers may not completely understood them because they may consist of a large number of lines of codes.

Such policy implementation is very specific to the codes of components in the system. In addition, a policy implemented in this manner would be very unstable with respect to the evolution of the system, because it can be violated by a change in the code of any member of the community. Finally, since even a simple modification may result in many inter-related changes to a variety of software artifacts, people are not willing to adopt intrusive approaches.

Current approaches to security requirements of composite services concentrates only on the concern from clients. While addressing the client concern, a number of projects only focus on monitoring quality of services, which mainly measure technical aspect of services, e.g. response time, availability; on workflow adaption, e.g. by switching to orchestrated services which provide the same functionality to improve the availability of composite services; and on enforcement constraints only on the client-orchestrator interaction. We believe in the context of service composition, besides addressing the client-orchestrator interactions, there is a need to deal with client concerns about the interaction between orchestrators and orchestrated services. Finally, we are not aware of any project which considers concerns of orchestrated servers.

For choreography scheme, while the technologies for implementing and interconnecting services under SOA are reaching a good level of maturity, modelling choreographies is still a big challenge. The problem is that, current choreography model specify choreographies by focusing on procedural aspects, leading to over-constrained solutions. It is important to note that the implicit assumption behind procedural models is that *anything that is not explicitly defined is not allowed to occur*. Thus, a choreography modeler must list all allowed executions when using those choreography specifications. Finally, there is no enforcement mechanism for choreography as they are currently modelled.

1.3 Our main approach and Dissertation Goal

It is our thesis that these difficulties can be alleviated by a suitable governance of the interaction between the disparate actors of a system, and without assuming any knowledge of, or control over, the structure or internal behavior of the interacting actors themselves. The thesis is based on the observation that much of the information that a system relies on involves the exchange of messages between the distributed components of the system.

One may doubt the usefulness of the governance that regulates only the interactions between the actors (or components) of the system being governed. But the critical role that essentially such governance plays in societal systems, suggests that it could be similarly useful when applied to software systems as well. The transportation system, for example, is governed by traffic laws that regulate such things as: to whom should one give the right of way, and

how should one react to the changing traffic lights. Such laws are oblivious of the intentions of individual drivers, yet they enable drivers to coordinate harmoniously with each other and negotiate their passage through intersections with relative safety. This, despite the fact that the drivers themselves are heterogeneous actors, with little if any knowledge of each other.

The overarching goal of my dissertation is to design a regulatory mechanism to address all these problems in a scalable manner. The mechanism is an extension of Law Governed Interaction (LGI) - a decentralized coordination and control mechanism for distributed systems. We call this mechanism ”LG-SOA”, for Law-Governed-SOA, which enables high expressive power, efficient enforcement, as well as good scalability. Our case studies demonstrate the flexibility and applicability of the mechanism. We did a series of experiments to show the overhead introduced by LG-SOA is relatively small, especially in the context of geographically distributed systems like SOA-based systems.

1.4 Thesis Plan

This thesis includes 10 chapters. In chapter 2, we present an overview of standard LGI. In chapter 3, we discuss the limitations of SOA-based systems. In chapter 4, we present a mechanism to address those limitations and discuss how it can be applied to legacy systems. In chapter 5, we describe an example of global constraints and its implementation. In chapter 6, we present server laws and their implementations. In chapter 7, we introduce orchestration scheme. In chapter 8, we discuss in more detailed problems of current choreography specifications, our approach to model choreography, and how we extend LGI to support choreography scheme. In chapter 9, we present a performance evaluation of our mechanism. Finally, we conclude in chapter 10.

Chapter 2

An Overview of LGI

In this chapter, we provide an overview of Law-Governed Interaction (LGI). This chapter is adapted based on LGI manual [8].

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 [3][5]. 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 [33] [37][51]. 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 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

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

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 [7].

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. 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 the home agent.

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 ⟨event⟩ if ⟨condition⟩ do ⟨action⟩
```

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 for expressing laws in LGI are Prolog and 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.

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, 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 [29].

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. 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 [18] 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 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.

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 [2].

---

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

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 [24][30] [51].

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 [28].

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)`auto-nomy:` 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 [2].

2.3.5 The Hierarchical Organization of Laws

LGI provides a mechanism to organize the laws into *hierarchies* [19][28]. 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.

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.

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. 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

The Limitations of SOA

3.1 Global constraints

SOA-based systems are heterogenous in the sense that services may be written in different programming languages, may run on different platforms, and may be maintained under different administrative domains. Moreover, services may be changed dynamically, or removed, while new services may be added in an unpredictable manner.

The heterogeneity of services in SOA-based systems calls for a means to impose global constraints on all service providers (servers). Global constraints may include properties such as the number of messages that can be sent by any given services; and operations such as stop, which would stop the service from servicing requests from its clients. There are diverse reasons for the existence of global constraints, including: (1) monitor/audit ability of the system, (2) control capability of certain behaviors of servers, (3) software engineering principles, (4) government regulations, etc. One of the objectives of this work is to enable the system as a whole to impose global constraints on the interactive behavior of the various heterogeneous servers. This would provide a certain level of uniformity across heterogeneous servers, despite different and specific functionality provided by those servers.

In particular, we will address two critical elements of management of global constraints: monitor/audit and control. Specifically, we provide system authorities, e.g system administrators or accountants, the capability to monitor/audit servers' performance and the power to control certain servers' activities. Such monitor/audit/control processes must be enforced because of two reasons. First, the data collected from the monitor/audit process must be *accurate* because system authorities make decisions based on them. Second, since servers *obey* control commands issued by system authorities, the system authorities can steer servers towards a goal of their choice.

3.2 Client-Server Interaction

Besides global constraints, a server may specify its own policies in which the server makes *specific* requirements and/or commitments regarding the services it is providing. There are several well-known incidents in which servers have violated their stated privacy policies. For example, in August 1998, the website Geocities released its users' personal data to advertisers [4]. Another example, in August 2006, the search queries of around 650,000 users were disclosed [5] with the intention of being used by third-party researchers [6]. There are possibly many more violations that are still not known by the public.

However, it is not just a matter of privacy. The interaction between a server and a client may involve a sequence of messages exchanged in a predefined order, called a conversation [3]. In this paper, we consider two types of server commitments. The first type is announced *before* the client-server interaction occurs. Privacy assurance is one commitment which belongs to this type. The second type of server commitments is made by the server *during* a conversation. For example, a service provider promises to sell a ticket to a client at price p if the client pays within an agreeable period g, say 3 days after the client reserves the ticket. Since the terms (e.g., p and g in the example just mentioned) exchanged between the server and the client are *unknown before* their conversation starts, there is a need to record those commitments and enforce them.

Note that even though a service provider has the freedom to specify its own commitments with clients in its server policy, the server policy must conform to the global constraints. Such conformance must be strictly enforced to ensure the imposition of global constraints across all servers in the system. We will present how we enforce this conformance in chapter 6.

It is hard or even impossible for clients to determine whether commitments made by servers will be satisfied. Therefore, clients need verifiable confidence that servers will abide by their commitments, provided that these commitments conform to the global constraints imposed by the system at large. However, clients cannot assume that service providers always abide by their policies because servers may sometimes choose to act against their commitments to gain more benefits. In other words, these global constraints and commitments must be enforced out of the control of servers.

There are a number of research projects attempting to enforce policies on interactions between components in a system. Unfortunately, they require the modification of existing applications. Such approaches are not widely adopted because of two reasons. First, even a simple modification may result in many inter-related changes to a variety of software artifacts. Consequently, re-implementation and retest of servers program are required, thus it would take long time to deploy these approaches. Second, system managers–who are responsible for implementing policies–have little, if any, sway over server programs, which may be dispersed all over the Internet, and whose often heterogeneous code may not be available, or even known, to the managers.

A server in SOA systems may not only interact with clients but also with other servers. In particular, the server can also interact with its peer servers under orchestration or choreography scheme. Orchestration describes how services can interact with each other from the *private* perspective and under control of a single service, which plays the role of an orchestrator. Choreography is associated with the *public* global visible message exchanges, rules of interactions and agreements that occur between multiple services. Such interactions may also pose certain problems regarding the integrity of the SOA systems, as we shall see below.

3.3 Interaction under orchestration scheme

Under orchestration scheme, a server, called orchestrator, orchestrates several servers to satisfy requests received from its clients. All interactions in the orchestration are conducted via the orchestrator. That means, the orchestrator can intercept, analyze and control the flow of messages exchanged between clients and orchestrated servers. As a result, clients and orchestrated servers may not know each other even though clients use services provided by these servers. Such indirect interaction may result in concerns for both clients and orchestrated servers.

There are four types of concerns coming from a client. First, when an orchestrator receives a request from a client, it can be considered as the client representative, and starts acting on the client behalf. Therefore, the client would want the orchestrator not to abuse that privilege. Second, typically a client has no knowledge about how/when an orchestrator interacts with its orchestrated servers to serve the client request. It is hard or even impossible for clients

to determine whether an action has been completed by the orchestrator or not. The client can eventually find out the orchestrator's misbehavior, but the consequence has been made. Therefore, the client would want to make sure that the orchestrator has actually completed certain things for the client, as stated by the orchestrator.

Third, the execution of a composite service depends on those of orchestrated servers. It may be possible that a client of a composite service expects the orchestrator only use service that comply with certain predefined requirements. Indeed, this expectation is just similar to that coming from clients in real life. For example, when a client buys a car, he/she would want that the car company only utilizes safe and standard-compliance components provided by certified providers. Fourth, clients may also need to know certain information regarding the actual interaction between the orchestrator and orchestrated servers. For example, assume an orchestrator receives commission proportioned to the total amount of money paid to all orchestrated servers. Consequently, the client would want the orchestrator to inform the client about the actual values of all purchases between the orchestrator and orchestrated servers required for the client request.

We would also consider two types of concerns from orchestrated servers. First, servers may choose to consider orchestrators as a special type of clients, and both the orchestrators and servers can receive benefits from that relationship. In particular, there is possibly a competition between equal servers in the sense that they provide same services as well as benefit to the orchestrator. How an orchestrator determines equal servers depends on business goal of the orchestrator, and is not discussed here. If this is the case, they would want the orchestrator to not favor some servers, and ignore other servers. Second, orchestrated servers do not directly interact with clients, and have to rely on data forwarded from orchestrators to process a request. Therefore, they need the cooperation of orchestrators in providing correct and necessary information about the clients.

3.4 Interaction under choreography scheme

While the technologies for implementing and interconnecting basic services under SOA are reaching a good level of maturity, modeling choreographies is still a big challenge. The problem

is that, current choreography models specify choreographies by focusing on procedural aspects, leading to over-constrained choreographies. Moreover, a choreography description in those models is not executable, i.e., there is no enforcement mechanism. Since a single misbehavior of an individual service may destroy the collaboration in a choreography, enforcement is a must to ensure safe and correct coordination between interacting services.

To overcome these limits, we propose that choreography models should focus on specifying the constraints required to make all services collaborate correctly, without stating how such a collaboration is concretely carried out. We believe that services should have the freedom to operate as well as they can, but they must satisfy the choreography constraints through enforcement. Thus, we also presente a mechanism to enforce choreography constraints.

Chapter 4

A Model for LG-SOA

In this chapter, we will present how we design a regulatory mechanism to address all problems discussed in chapter 3. The mechanism is an extension of Law Governed Interaction (LGI). We call this mechanism LG-SOA, for Law-Governed-SOA, which enables high expressive power, efficient enforcement, as well as good scalability.

Broadly speaking, an LG-SOA system S is defined as a triple (B, LE, T), where B-the base system-is the set of actors that constitute the system to be governed; LE-the law ensemble-is the conformance hierarchy of laws that collectively govern the interactive activities by the components of B; and T is a set of generic LGI controllers that serve as the middleware that mediates the interactive activities of the components of B, subject to the various laws in LE.

We assume no knowledge of the structure and behavior of the actors of B. But for specificity we will assume that all these actors are software components- although, in fact, some of them may, for example, be people communicating via their smart phones. The only substantial assumption we make in this section about the components of B, is that they all communicate via the LGI middleware, subject to laws in LE. But, as we will show later, even this assumption can, and should, be relaxed somewhat.

4.1 The Anatomy of LG-SOA based systems

Figure 4.1 describes a system schematically consisting of two levels: (a) the lower level, which consists of the set of components of the base system B; and (b) the upper level, which consists of the set T of LGI-controllers that mediates the interactive activities of the components of B, thus enforcing the various laws that belong to the law-ensemble LE. Above these two levels, this figure depicts some actors that do not belong to the system at hand, but interact with components in B by exchanging messages with them, as shown in the cases of u and v.

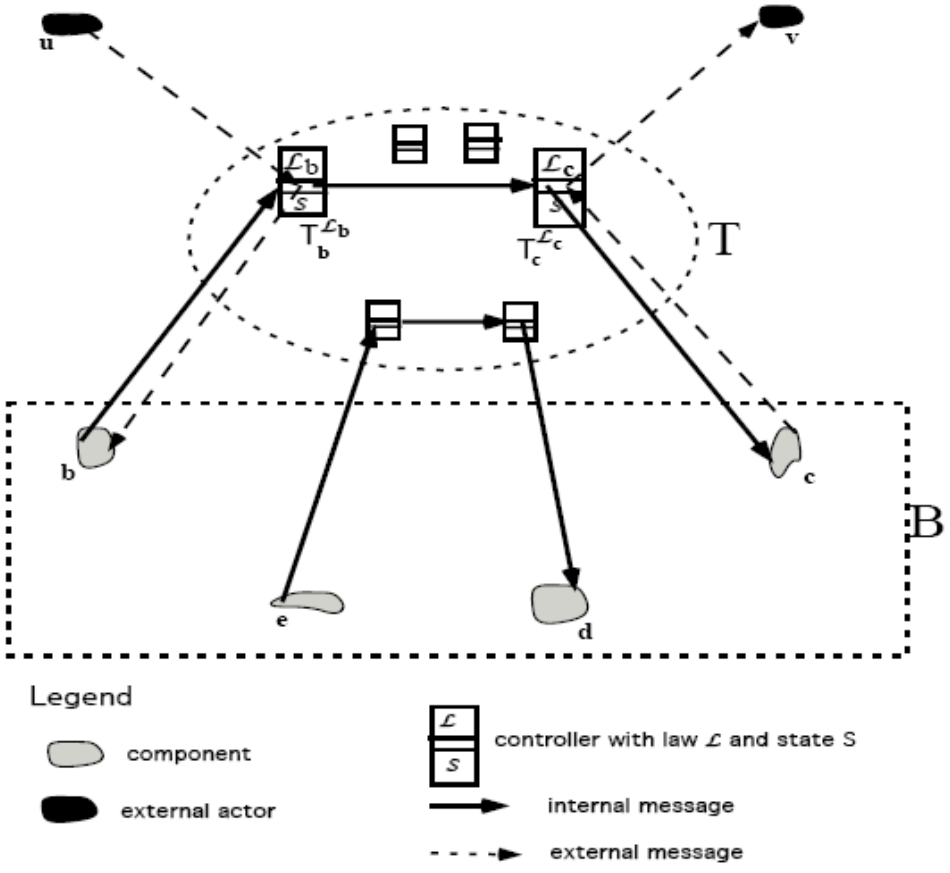

Figure 4.1: A Model for Law-Governed Service Oriented Architecture (LG-SOA)

The controllers in T belong to what we have called the controller service, or CoS, and constitute the distributed trusted computing base (DTCB) of this architecture. Every component x in B is paired with a controller $T_X^{\mathcal{L}}$, that mediates all messages sent or received by x, subject to the law $\mathcal{L}$ under which x operates. Here $\mathcal{L}$ may be what we call the component law designed specifically for x, as in the case of component b in figure 4.1, which is associated with controller $T_b^{\mathcal{L}_b}$ that operates under law $\mathcal{L}_b$. But x may also operate under one of the division laws of the hierarchy, or even under its root law. Specifically, it is possible that a server does not specify its own law and chooses to operate under global law while several others servers are regulated by the same server law.

Note that a message exchanged between a pair of components of the base system B, say b and c, is mediated by a pair of controllers $T_b^{\mathcal{L}_b}$ and $T_c^{\mathcal{L}_c}$ (assuming that both components operate under their own laws)-a case of the dual mediation under LGI. On the other hand, the

interaction of a component, say c, of B with external actors, such as v, may be mediated by controller $T_c^{\mathcal{L}_c}$ alone, because external actors may not operate under LGI. For example, client-server interaction is only regulated by the controller of the server.

4.2 Seamless Interoperation

The term "interoperation" means the exchange of messages between actors operating under different laws (or policies - Note that this subject has been studied extensively under access control, that uses the term "policy" for what we call here "laws".), say $\mathcal{L}$ and $\mathcal{L}'$. This model require massive interoperation, because it is very common for a pair of interacting components, such as b and c in figure 4.1, to operate under different laws, although all these laws belong to the same hierarchy.

The conventional treatment–under access-control–of interoperation [47, 48, 49, 50, 51] calls for formation of a composition of the two policies, which is consistent with both of them, and then mediates the interaction under this composite law, by means of a single mediator. Unfortunately, as shown in [49], such compositions tend to be computationally hard, even for the simple types of policies used for access control–and it is often the case that no composition exists because the laws to be composed are not compatible. Moreover, if a system has N interacting components, each operating under its own law, one may have to create O(N^2) such compositions–quite an impractical proposition.

Fortunately, as we have shown in [1], when the laws under which the interacting components operate belong to the same conformance hierarchy, and if they are "willing" to interact if both conform to a common law (such as the global law $\mathcal{L}_\mathcal{G}$ in our case) then the interoperation between them may be completely seamless, requiring no composition. This is due to two factors. The first is the dual control over every message exchange between components in B, where the law of each interlocutor is enforced by its own mediator. This is different from traditional policy-mechanisms, under which interaction between a pair of actors is always mediated by a single mediator. The second factor that allows seamless interoperability is the nature of conformance hierarchy, which provide the assurance to the two interlocutors, that both operate under the same superior root law. In our case, for example, component c can be confident that

the message it get from b contains the correct identification of it-because such identification is required by law $\mathcal{L}_{\mathcal{G}}$.

4.3 The Sense in which Laws are Enforced Under LG-SOA

A given hierarchical law-ensemble LE is clearly enforced under LG-SOA, if all the components of a given system exchange their messages via LGI controllers, under laws belonging to the LE of that system, as described above. But how can one be sure that this condition is satisfied? The problem is that it is quite impossible to force a set of disparate components, dispersed throughout the Internet, to conform to any given constraint on their communication, beyond the constraints implicit in the standard Internet protocols. (It is worth pointing out, however, that it is possible to do so over an Intranet. In particular, as has been demonstrated in [46], this can be done by having the collection of firewalls that mediate all Intranet communication, rerouting messages to appropriate LGI-controllers.)

Nevertheless, one may often be virtually compelled to operate under a particular law $\mathcal{L}$, or under some law belonging to a specific hierarchical law ensemble. Broadly, this is the case when one wishes to communicate with others that require their interlocutors to operate under a given law $\mathcal{L}$, or under any subordinate law to $\mathcal{L}_{\mathcal{G}}$.

For client-server interaction, global law and server law are enforced only if clients send requests to server's controller. But, why do clients choose to do this? That happens because of two reasons. First, only registered services are maintained in a system registry, which is responsible for maintaining information of all servers in the system. Since the registry provides clients with location of server's controller, clients may not know the $real$ physical address of a server and have to send requests to server's controller. Second, even if client know the real address of the server, the client may still send request to the server's controller because clients may want assurance that server commitments are enforced. Note that if server commitments are enforced, global constraints are enforced.

At the server's side, it is possible that servers require clients send requests via their controllers, and would not process requests sent directly from clients. Servers impose this requirement because it gain benefits by doing so. For example, some servers may want to prove its

usefulness, e.g., serving many requests, to system managers. However, since the managers trust controllers but not those servers, only performance reports from controllers are accepted.

If no one has interest in forcing clients to send requests via server controllers, we face a common problem: we cannot force someone to do something if they do not get benefit by doing this.

For server-server interaction, this is so because a given LGI-agent can detect if its interlocutor is an LGI-agent as well, and can identify the law under which it operates. For a more concrete demonstration, consider our case study whose root law $\mathcal{L}_{\mathcal{G}}$ contains-among others-the provision of service invocation. And suppose that the budget office does its communication subject to law $\mathcal{L}_{\mathcal{G}}$, including assigning budgets to various components, and obtaining income reports from servers.

Now, assuming that servers get a substantial credit for the income they report to the budget office, they will have to do the reporting under law $\mathcal{L}_{\mathcal{G}}$ (or under a subordinate law), given that this office accept such reports only subject to this law. This, in turns, means that servers would insist to receive service request only under $\mathcal{L}_{\mathcal{G}}$, because this would allow them to accumulate their income in the state of their controller operating under $\mathcal{L}_{\mathcal{G}}$, which can then be sent to the budget office. And this, finally, means that service requests must be sent via a controller that operates under law $\mathcal{L}_{\mathcal{G}}$, and maintains its budget assigned to it by the budget office. So, in conclusion, if the budget office operates under $\mathcal{L}_{\mathcal{G}}$ (or under a subordinate law), then both the servers and their clients would be effectively forced to operate under $\mathcal{L}_{\mathcal{G}}$, or under a subordinate law–at least for participating in the service ordering activity.

Due to such needs to operate under certain laws, one is reasonably justified to claim that the laws of a given LE are enforced under LG-SOA. And it is worth pointing out that such, somewhat loose, usage of the term "enforcement" is quite common in the access-control literature. For example, the XACML mechanism [19] relies for its "enforcement" on each server within its purview to voluntarily operate via a *policy enforcement point* (PEP). This PEP is expected to consult a *policy decision point* (PDP) regarding every request sent to it, and then to carry out the decision of the PDP. But no means are provided for actually forcing servers to operate via such PEPs, and no such enforcement is possible in an open system, as explained above. In fact, the trustworthiness of enforcement under the LG-SOA mechanism is stronger than under

conventional access control, due to the inherent trustworthiness of the generic controllers, and because they can recognize each other across the Internet.

More scenarios in which one may often be virtually compelled to operate under a particular law $\mathcal{L}$ are presented in chapters 7 and 8.

Strict Enforcement Of Lenient Laws:

Note that LGI laws are enforced strictly, in that a system governed by a given law is prevented from violating it—subject to the qualification made above. However, the law itself may be lenient. For example, instead of having a law that restricts certain kind of messages—which would block them if they are sent—one can write a law that does not block such undesirable message, but only logs them, so they can be reported later to those that can change the code that sent these messages. Such lenient laws are important for subjecting legacy system to LG-SOA.

4.4 A concrete setting of SOA-based system

Let us take a closer look at a setting of SOA-based system.

Figure 4.2 gives an overview of a system which consists of a set of servers SP_1, SP_2, ..., SP_m, a set of controller T_1, T_2, ..., T_m, a set of clients C_1, C_2, ..., C_1, one service registry, CA, and one law server. The CA, certification authority, is responsible for issuing certificates. The registry is used by service consumers to find services that match certain criteria. If the registry has such a service, it provides the consumer with a location address for that service. Each server SP_i is assigned a controller, called T_i, which is responsible for enforcing server law $\mathcal{L}_i$.

It is possible that a server does not specify its own law, and chooses to operate under global law, as in the case of SP_k. It is also possible that several servers are subject to the same server laws, as in the case of SP_1 and SP_2, both of them are subject to law $\mathcal{L}_i$.

We employ 2-level law hierarchy: the first level of the hierarchy is for the global law, and the second is for server laws, orchestration laws and choreography laws. As shown in figure 4.2: $\mathcal{L}_1$ ... $\mathcal{L}_m$ are server laws, $\mathcal{L}_{O_1}$ ... $\mathcal{L}_{O_t}$ are orchestration laws, and $\mathcal{L}_{C_1}$ ... $\mathcal{L}_{C_Z}$ are choreography laws. They are direct subordinates of the global law $\mathcal{L}_G$.

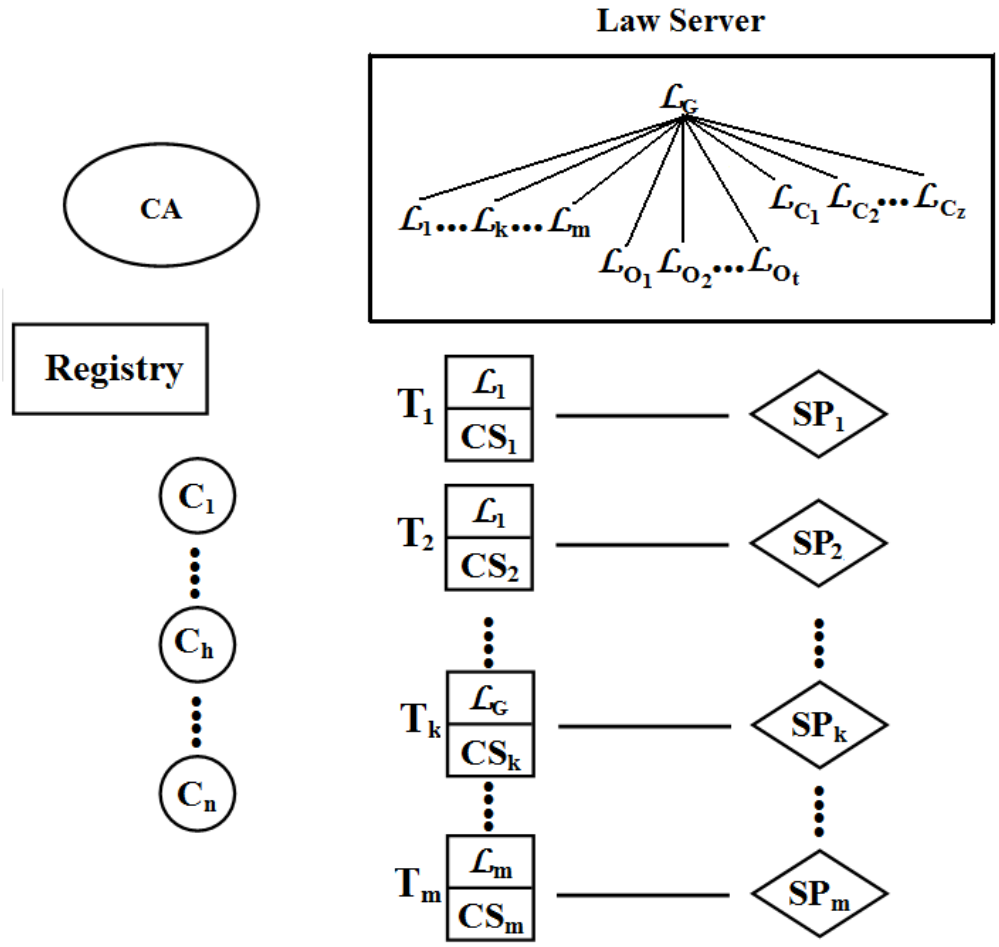

Figure 4.2: SOA-based system - A concrete setting

LGI hierarchy provides service providers with the freedom to specify their own laws, while it still enforces the conformance of server/orchestration/choreography laws to global law. The enforcement process is shown in figure 4.3.

In figure 4.3, $\mathcal{L}$ is directly subordinate to $\mathcal{L}_G$, so law evaluation starts in $\mathcal{L}_G$. This way, the constraints imposed by $\mathcal{L}_G$ are included in the ruling of $\mathcal{L}$. In other words, the conformance of $\mathcal{L}$ to $\mathcal{L}_G$ is enforced.

If a delegate statement is met in $\mathcal{L}_G$, $\mathcal{L}$ is consulted and a refinement is proposed by $\mathcal{L}$. In essence, this mechanism provides $\mathcal{L}$'s law maker with an opportunity to specify additional rulings. We will see how SP_i refines global constraints in chapter 6. Finally, $\mathcal{L}_G$ can rewrite $\mathcal{L}$'s refinement if it sees fit.

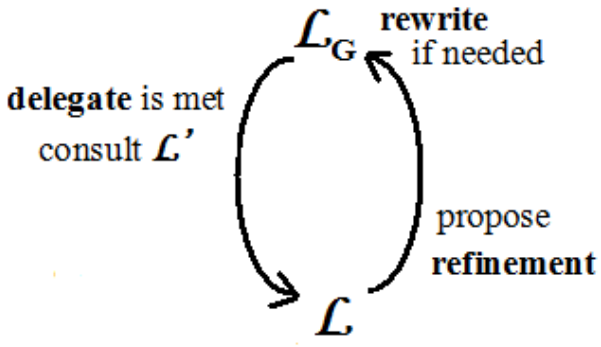

Figure 4.3: Law Conformance Enforcement

4.5 Controller management interfaces

Controllers are *generic* in the sense that they are capable of enforcing any law. Initially, a generic controller T operates under a generic law called $\mathcal{L}_{gen}$. Generally speaking, the "generic" law defines who can use the set of management interfaces to manage controllers. How system authorities decide the constraints in $\mathcal{L}_{gen}$ varies depends on specific scenarios. We, however, will present an example of the generic law shortly later for demonstration purposes.

Below we consider two important interfaces in *set of management interfaces*.

1. *grant*(*SPname, endPoint, lawURL*): upon receiving a "grant" message, the generic controller T verifies if this message is issued by a right system authority as defined in $\mathcal{L}_{gen}$. If this is the case, then T becomes T_i to enforce the law of the server whose name is "*SPname*" and physical address is "*endPoint*". The law text will be retrieved from the law server based on *lawURL.*

2. *revoke()*: similarly, if T_i receives a "revoke" message from SA, T_i is back as a generic controller operating under $\mathcal{L}_{gen}$.

Note that $\mathcal{L}_i$, in turn, defines how the management interfaces can be invoked. Since $\mathcal{L}_i$ conforms to $\mathcal{L}_g$, we can prevent the management interfaces from being misused in $\mathcal{L}_i$ by specifying in $\mathcal{L}_g$ who can invoke what interfaces under which conditions. For now, we only implemented these two interfaces (grant/revoke). Other interfaces can be added if necessary in the future.

Having the capability to manage the controller through management interfaces, system authorities have the power to manage all servers in the system.

4.6 The implementation of Laws $\mathcal{L}_{gen}$

Law $\mathcal{L}_{gen}$ shown in figure 4.4 is just for demonstration purpose. It can be more complex in real scenarios.

Preamble:
law $\mathcal{L}_{gen}$
authority(CA, keyHash)

$\mathcal{R}$1. **U**PON arrived(_,[reqID, s, reqcnt],self) DO
cert = SearchST("Cert", reqcnt)
IF(cert[issuer(CA),subject(SA)])DO
IF (s=="grant")
{
SPName = SearchST("SPName", reqcnt)
EP = SearchST("EndPoint", reqcnt)
lawURL = SearchST("lawURL", reqcnt)
grant(SPName, EP, lawURL)
}

Figure 4.4: Law $\mathcal{L}_{gen}$ - Generic Law

Law $\mathcal{L}_{gen}$'s "preamble" specifies that $\mathcal{L}_{gen}$ accepts certificates from CA represented by keyHash. $\mathcal{L}_{gen}$ is very simple, and it has only one rule. This rule specifies that a generic controller T has to stop working under the generic law $\mathcal{L}_{gen}$ if T receives a $grant$ command from SA. The "grant(SPName,EP,lawURL) signifies that the controller should be ready to enforce the law of a server whose name is $SPName$, endpoint is EP, and law text is available at $lawURL$.

4.7 Grant/Revoke Server Privilege

To join the system, first SP_i requests that SA do two things: (1) appoint an available generic controller T to become T_i, i.e. to enforce $\mathcal{L}_i$, and (2) publish SP_i's service description to the UDDI server *using T_i's address as the service's endpoint*. This way, a client who wants to invoke SP_i's services sends its requests to T_i, and gets a corresponding response from T_i.

In the SP_i's service description published to the UDDI server, SA also *records the URL*

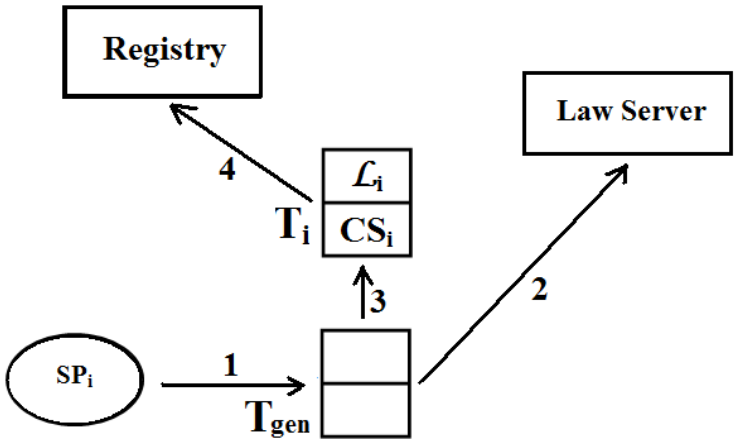

Figure 4.5: A server joins the system

of $\mathcal{L}_i$ (i.e. SP_i's law URL). Currently, we have this value saved in the optional field named *description* of SP_i's *businessEntity* structure. In fact, we can save the URL of $\mathcal{L}_i$ in *any optional element* of SP_i's service description.

A client can query the UDDI server about SP_i's law URL, based on which it can examine the law text (maintained by the law server). Moreover, since T_i is out of the control of SP_i, the client can be confident that $\mathcal{L}_i$ will be strictly enforced (i.e. SP_i's commitments will be satisfied).

4.8 T_i handles a client's request

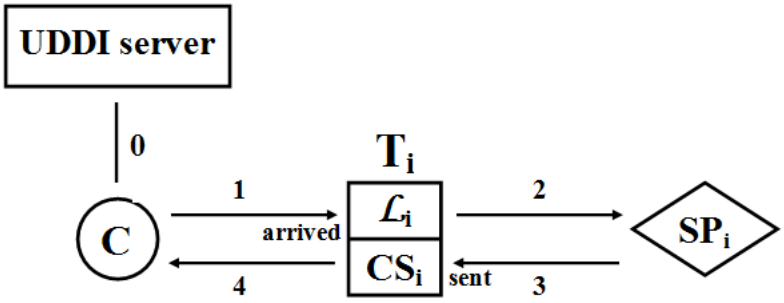

Figure 4.6: T_i handles a client's request

The interaction between a client c, T_i, and SP_i is presented in figure 4.6. In step 0, c contacts the UDDI server to get SP_i's service description. c only needs to query the UDDI server *once* to get SP_i's service description the first time it wants to use SP_i's services.

In step 1, based on SP_i's service description, c creates a SOAP message to invoke a desired service. It establishes a connection, $conn_1$, to send a request to T_i. T_i accepts and maintains $conn_1$ through which T_i can return the response back to c. T_i generates a unique identifier, $reqID$, to identify the request. The arrival of m to T_i leads to the firing of an **arrived** event at T_i. T_i evaluates the ruling of the law $\mathcal{L}_i$ for the **arrived** event, and produces a list of operations to be carried out. If that list contains an operation *forwardRequest(m)*, T_i will forward m to SP_i. Note that m may be c's original request, or a modified version of the original message, or a newly-created one.

In step 2, T_i acts as a client of SP_i. It initiates a connection, $conn_2$, to send m to SP_i. SP_i processes the request m, and returns the response through $conn_2$ in step 3. As soon as T_i receives the response, a **sent** event is fired at T_i. Similar to step 1, the **sent** event is evaluated, and a list of operations to be carried out is generated. If this list contains an operation *forwardResponse(m)*, m will be returned to c through $conn_1$ in step 4.

Since T_i associates $conn_1$ with $conn_2$ using the same $reqID$ mentioned above, and since $conn_1$ is kept until c receives the response, our mechanism supports asynchronous service invocations. Finally, since c's request is not sent directly from c, SP_i cannot locate c from $conn_2$. Therefore, c's location (host, port, etc.) is *inherently hidden* from SP_i.

The security infrastructure is comprised of SA, CA, $Acct$, law server, UDDI server, and controllers. One important note to make is that the process of appointing a controller to enforce a server's law does not require any change to the applications of the server, the client, and the UDDI server - three main components in SOA systems. Moreover, their normal behaviors are preserved: (1) the UDDI server allows authorized entities (SA, as assumed here) to update server descriptions, (2) the client queries the UDDI server to get the server descriptions, (3) the controller plays the role of the server when interacting with the client, (4) the controller acts as a client of the server. Therefore, we believe that our approach is easily integrated into any existing SOA system.

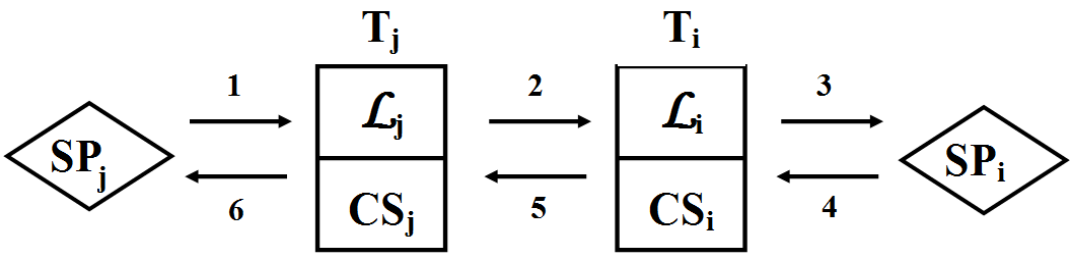

Figure 4.7: Dual mediation

4.9 Dual mediation

Figure 4.7 explains how a request sent from a server SP_j to another server SP_i is treated, as well as how the response from SP_i is sent back to SP_j.

Each time a server SP_j needs to invoke a service provided by SP_i, SP_j creates a SOAP message according to SP_i's service description retrieved from the UDDI server. SP_j also updates the header of the SOAP message to indicate that the final receiver of the SOAP message is T_i. Here, we use the *SOAP intermediary* feature described in the SOAP message header. In step 1, SP_j sends the SOAP request to its controller T_j which in turn decides if SP_j is allowed to send such a message according to $\mathcal{L}_j$. If this is the case, in step 2, T_j removes the SOAP intermediary field from the header of the SOAP request, and forwards it to T_i. Finally, T_i forwards the request to SP_i after evaluating the message according to $\mathcal{L}_i$ in step 3. The SOAP response is sent back to SP_j using the same path but in reverse order. The SOAP response is also mediated by $\mathcal{L}_i$, and then $\mathcal{L}_j$, respectively.

In this interaction, a SOAP request or response is regulated by both T_j and T_i, hence the terminology dual-mediation. This technique can be used for an orchestrator to invoke a service provided by orchestrated servers.

Chapter 5

Global Law

In this section, we will present an example of global constraints, and its implementation.

5.1 Description of Global Constraints

Suppose we want to impose the following global constraints on a specific system consisting of multiple servers:

1. **Performance Monitor**: each server must report its performance to $Acct$ periodically. For example, a server must report its average response time (ART) every 60 minutes.

2. **Interaction Audit**: Each server must provide audit information about interactions with its clients if requested by $Acct$. This constraint provides a way for $Acct$ to examine a server's activities in more detail, if necessary. An example of this constraint is:

a. Upon receiving an *"initAudit"* message from $Acct$, a server must immediately start reporting any requests it receives to $Acct$. The audit information of a request includes the name of the service to be invoked, the time the request is received, and the time the corresponding result is returned.

b. Upon receiving an *"endAudit"* message from $Acct$, the server stops reporting audit information to $Acct$.

3. **Control Ability**: Each server must obey *control* commands issued by SA. This provides a way for SA to control certain server activities. Below is an example of this constraint:

a. Upon receiving a command *"stop"* from SA, a server must respond to any client's request with the message "Server is stopped".

b. Upon receiving a command *"resume"*, the server resumes its normal operation.

These global constrains are specified in a policy called P_g (for Global Policy). Even though

P_g has been deliberately made simple for ease of presentation, it still demonstrates various important aspects inside the system. First, P_g imposes global constraints on all servers. Specifically, SA and $Acct$ together can monitor/audit/control certain server activities. Second, P_g implies that a history of interactions between a server and its clients can be maintained. For example, ART is a value reflecting one aspect of previous interactions. Below we will further investigate stateful policies illustrated in the form of server policies. Finally, P_g does not meddle with any specific server-client interaction, leaving them for the individual server to specify.

5.2 Implementation

Services that utilize Web services standards (e.g., Web Service Description Language (WSDL), Simple Object Access Protocol (SOAP)) are the most popular type of services available today. A server and its clients communicate via messages that are formatted according to the Simple Object Access Protocol (SOAP). Before the server is deployed and starts interacting with its clients, the server's manager decides the structures of the SOAP requests and responses in its WSDL file. Therefore, typically the server can take *the specific format of the SOAP request/response into account when it specifies its law*, as we will see in $\overline{\mathcal{L}_2}$ and $\overline{\mathcal{L}_3}$.

There is an *implicit* constraint in $\mathcal{L}_g$. That is, for each command in the set $\{initAudit$, $endAudit$, $stop$, $resume\}$ sent to T_i, the SOAP request has a field *Cert* whose value contains the certificate of the one who sends the command. T_i verifies that certificate to determine if the command has been issued by the right authority.

Law $\mathcal{L}_g$ is shown in figure 5.1. Its preamble specifies that $\mathcal{L}_g$ is a root law, and accepts certificates from CA represented by keyHash. The control state has four protected terms *'inv(req(reqID), act(s), T(t))'*, *'numOfInv(num)' 'totalPT(pt)'*, and *'Acct(host(h), port(p))'*. They are used to record the time t the controller T_i receives a request $reqID$ to invoke a service s of SP_i, the number of requests processed by SP_i so far, the total processing time, and the (host/port) of $Acct$'s application, respectively.

In *arrived/sent* events, the parameters **reqcnt/respcnt** are the **original** client's SOAP request and SP_i's SOAP response, respectively. The parameter **reqID** is used to identify a particular client's request and its associated response. The *client's certificate* **cert**, and *the name*

Preamble:

law $\mathcal{L}_g$
authority(CA, keyHash)
protected(Inv(req(_),ser(_),t(_)))
protected(numOfInv(_))
protected(totalPT(_))
protected(Acct(host(_), port(_)))

$\mathcal{R}1$.
```
UPON adopted() DO
        imposeObligation("reportART", 60 x 60)
        add(Acct(host(host), port(port)))
```

$\mathcal{R}2$.
```
UPON arrived(_, [reqID, s, reqcnt], _) DO
        IF (s @ (command set of Lg)) DO
        {
         cert = SearchST("Cert", reqcnt)
         IF (cert[issuer(CA),subject(Acct)]) DO
         {
           IF (s =="initAudit") DO add(audit)
           IF (s =="endAudit") DO remove(audit)
         }
         ELSE IF (cert[issuer(CA),subject(SA)])  DO
         {
           IF (s =="stop") DO add(stop)
           IF (s =="resume") DO remove(stop)
         }
        }
        ELSE DO // a normal client's request
        {
         IF ("stop"@CS) DO    forwardResponse("Server is stopped")
         ELSE DO
         {
         add(Inv(req(reqID),act(s),T(curTime)))
         delegate(arrivedCont,[reqID,s,reqcnt])
         }
        }
```

$\mathcal{R}3$.
```
UPON sent(_, [reqID, respcnt], _)) DO
        inv(req(reqID),act(s),T(t)) @ CS
        t1=curTime,
        inc(totalPT,t1-t)
        inc(numOfInv,1)
        remove(inv(req(reqID),act(s),T(t)))
        IF ("audit"@CS) DO
        {
           Acct(host(h), port(p))@ CS
           release(Self, [s, t, t1], h, p)
        }
        delegate(sentCont)
```

$\mathcal{R}4$.
```
UPON obligationDue(obl) DO
        IF (obl == "ReportART") DO
        {
           Acct(host(h), port(p)) @ CS
           numOfInv(num)@CS
           totalPT(time)@CS
           release(Self, num/time, h, p)
           imposeObligation("ReportART",60x60)
        }
        delegate(obligationCont, obl)
```

Figure 5.1: Law $\mathcal{L}_g$ Global constraints

of the service **s** the client wants to invoke are extracted from **reqcnt**. **s**'s parameters and their corresponding values are in **reqcnt**.

Rule $\mathcal{R}_1$ is the specification of $\mathcal{L}_g$'s **adopted** event, the first event in the life of T_i. $\mathcal{R}_1$ imposes an obligation to have T_i report SP_i's ART to $Acct$ every hour. The $(host, port)$ of $Acct$'s application is also added to T_i's control state CS_i.

Rule $\mathcal{R}_2$, $\mathcal{L}_g$'s **arrived** event, is fired upon the arrival of a request to T_i. T_i first checks the request's type. If this is a command from SA or $Acct$, T_i updates CS_i accordingly, and the updated CS_i will affect T_i's behavior. If this is a client's request, and (1) if there is a term "stop" in CS_i, T_i returns a message "Server is stopped" back to the client; (2) otherwise, T_i records the request's information by adding a 3-tuple (reqID, act, t) to CS_i, and this **arrived** event is delegated to a law refinement $\overline{\mathcal{L}_i}$ for further rulings. Specifically, the *delegate* statement indicates that the goal **arrivedCont** is open for refinements.

Rule $\mathcal{R}_3$, $\mathcal{L}_g$'s **sent** event, occurs when the response of a request is sent back from SP_i. Based on *reqID* and the term *inv(req(reqID),ser(a),T(t))* in CS_i, T_i calculates the request's processing time, and updates CS_i. If SP_i is under an audit, i.e. there is a term "audit" in CS_i, T_i will report the request's auditing information to $Acct$. Like $\mathcal{R}_2$, $\mathcal{R}_3$ has a goal named **sentCont** which can be refined by an $\mathcal{L}_g$'s subordinate.

Rule $\mathcal{R}_4$ is discharged when "ReportART" comes due. This obligation requires T_i to report SP_i's ART to $Acct$ every 60 minutes. Then, "ReportART" is set to fire again in the next 60 minutes.

Chapter 6

Server Law

6.1 Server's law

A server's law can be considered as the server's contract with its clients. In that contract, the server makes *specific* requirements and/or commitments regarding the services it is providing. Below we will consider three types of useful server policies.

6.1.1 Law $\mathcal{L}_1$: Service Level Agreement (SLA)

Our first example is a simple case of SLA: If the time difference between the arrival of a request to the server and the return of the corresponding response from the server is greater than a predefined threshold, say 10 seconds, then the server will notify the accountant $Acct$ of the actual difference, and a copy of this notification will be sent to the client.

6.1.2 Law $\mathcal{L}_2$: Server promises during a conversation

The interaction between a server and a client may involve a sequence of messages exchanged in a predefined order, called conversation [3]. During their conversation, the server may make promises to the client, and must be accountable for them.

Suppose SP_2 is a travel agent which supports a "ticket selling" conversation. During such a conversation, a client c may request to reserve the right to buy a ticket at a particular price p with a grace period g. If SP_2 agrees, it is obliged to sell this ticket to c for p, if c pays for the ticket within period g. Moreover, SP_2 commits itself to not sell the reserved ticket to anybody else before the end of the grace period.

6.1.3 Law $\mathcal{L}_3$: Assurances of Privacy

Given a client's request, a server can determine the client's sensitive data which include (1) IP address of the client's application and (2) the client's identity extracted from the request's content. As soon as the server receives the client's request, the client no longer has control over its sensitive data. To assure clients that their sensitive data are not misused, the server announces its privacy policy, and must adhere to this policy.

Suppose that SP_3 also needs to *link messages belonging to the same conversation* as in the "travel agent" scenario above. Below is an example of SP_3's privacy law $\mathcal{L}_3$.

a. **Client Identification**: A client has to submit its certificate signed by CA to identify itself when it requests for SP_3's services.

b. **Privacy Assurance**: SP_3 will not release the client's certificate and IP address to other third parties.

6.1.4 Relationship between $\mathcal{L}_i$ and $\mathcal{L}_g$:

SP_i has the freedom to specify $\mathcal{L}_i$, but $\mathcal{L}_i$ must conform to $\mathcal{L}_g$. For example, it is not possible for SP_i to write $\mathcal{L}_i$ to not report its ART periodically to $Acct$, or to not obey SA's commands. Such provisions imposed by $\mathcal{L}_g$ on $\mathcal{L}_i$ are expressed through LGI law hierarchy which are strictly enforced by our mechanism, as explained in section 4.4.

For the sake of simplicity, we only present $\overline{\mathcal{L}_2}$ and $\overline{\mathcal{L}_3}$ in following sections. The implementation of $\overline{\mathcal{L}_1}$ is straightforward.

6.2 Server Law $\mathcal{L}_2$

A conversation between SP_2 and its client involves three services: "*asks*", "*reserves*", and "*pays*". The client c invokes the "*asks*" service to query SP_2 about available tickets (uniquely identified by *tkIDs*) and their corresponding prices. We view SP_2's response for the "asks" service as *just for the client's reference*. c can use service "*reserves(tkID, pp)*" to propose to buy the ticket *tkID* from SP_2 at an exact price *pp* if it makes a payment within a period of 20 minutes. Note that *pp* **can be different** from the ticket's price returned from the "asks" service. The controller T_2 does **not** influence SP_2 regarding how to deal with c's proposal. Therefore,

Preamble:
law $\mathcal{L}$ refines $\mathcal{L}_g$
authority(CA, keyHash)

$\mathcal{R}1$.
```
UPON arrivedcont([reqID, s, reqcnt]) DO
        add(Inv(req(reqID),act(s),T(curTime)))
        IF (s=="asks") DO forwardRequest(reqcnt)
        ELSE DO
        {
         id = searchST("TicketID", reqcnt)
         cert = searchST("Cert", reqcnt)
         IF (rs(tID(id),cl(_),tP(_),T(t))@CS) DO
         {
          -- cancel expired reservation of "id"
          IF (curTime - t > 20 x 60) DO
           remove(rs(tID(id),cl(_),tP(_),T(t)))
         }
         IF (s == "reserves") DO
         {
         IF (rs(tID(id),cl(_),tP(_),T(_))@CS) DO
            deliverResponse("Somebody else has
                        reserved this ticket.")
         ELSE IF (cert[issuer(CA),subject(c)]) DO
          {
            p=searchST("ProposedPrice",reqcnt)
            add(pr(req(reqID),tID(tkID),
                                  cl(c),tP(p)))
            // create or find in CS c's nickname
            n=findNick(c);
            reqcnt1=replaceST(n,"Cert",reqcnt)
            forwardRequest(reqcnt1)
          }
         }
         ELSE IF (act == "pays") DO
          IF (cert[issuer(CA), Subject(c)]) DO
          {
          IF !(rs(tID(id),cl(c),tP(p),T())@CS)   DO
           deliverResponse("Cannot pay for
                        a non-reserved ticket.")
          ELSE DO
          {
           t1 = curTime
           add(paid(tID(id),cl(c),tP(p),T(t1)))
           Acct(host(h), port(p))@ CS
           release(Self, [paid], h, p)
           // create or find in CS c's nickname
           n=findNick(c);
           reqcnt1=replaceST(n,"Cert",reqcnt)
           forwardRequest(reqcnt1)
          }
         }
        }
```

Figure 6.1: Refinement $\overline{\mathcal{L}_2}$

```
R2. UPON sentcont([reqID, respcnt]) DO
          inv(req(reqID),act(s),T(_))@CS
          IF (act == "reserves") DO
          {
           ans=searchST("ReservesResult",respcnt)
           IF (ans == "Reservation Accepted.") DO
           {
            pr(req(reqID),tID(tkID),cl(c),tP(p))@CS
            t= curTime
            add(rs(tID(tkID),cl(c),tP(p),T(t)))
            remove(pr(req(reqID),tID(tkID),
                                  cl(c),tP(p)))
           }
          }
          deliverResponse(respcnt)
```

Figure 6.2: Refinement $\overline{\mathcal{L}}$ (con't)

if SP_2 agrees, SP_2 must be **accountable for its decision**. c uses service "*pays(tkID)*" to pay for its reserved ticket.

$\overline{\mathcal{L}_2}$ is shown in figure 8. $\overline{\mathcal{L}_2}$'s preamble states it is a law refinement of $\mathcal{L}_g$. To record that client **c** successfully reserves ticket **tkID** for a price **p** at time **t**, T_2 saves a term *rs(tID(tkID),cl(c),tP(p),T(t))* in CS. Rule $\mathcal{R}_1$ refines the goal **arrivedCont** delegated from $\mathcal{L}_g$'s *arrived* event. When c "asks" SP_2 for available tickets, T_2 simply forwards c's inquiry to SP_2. In both "reserves" and "pays" cases, T_2 first cancels expired reservation of the involved ticket, if any. It does so by calling the function $searchST("TicketID", reqcnt)$ to extract the value *tkID* of the field *TicketID* in the SOAP request *reqcnt*. Then, T_2 removes any 4-tuple *rs(tkID, client, price, time)* of which the elapsed time *(curTime-time)* is greater than 20 minutes.

Consider the case when c uses "*reserves(tkID,pp)*". If there is a valid reservation for *tkID*, T_2 returns to c the message "Somebody else has reserved this ticket" and drops c's request. This way, T_2 *prevents a ticket from being reserved by more than one client at the same time*. If there is no reservation for *tkID* and if c presents a valid certificate, T_2 calls the function $searchST("ProposedPrice", reqcnt)$ to extract the value *pp* of the field *ProposedPrice* in c's SOAP request *reqcnt*. T_2 saves c's proposal by adding into CS the term $pr(req(reqID), tID(tkID), cl(c), tP(pp)$ (which will be used in rule $\mathcal{R}_2$ of $\overline{\mathcal{L}_2}$), and forwards c's request to SP_2.

Consider the case when c invokes "*pays(tkID)*". If c's certificate is valid and its reservation is still effective, T_2 will **sell the ticket to the client on SP_2's behalf**. Specifically, T_2 adds the

term $paid(tID(tkID), cl(c), tP(p), T(t1))$ into CS. This term serves as a **receipt** stating that **c** buys the ticket **tkID** at price **p** at time **t1**. Next, T_2 reports c's receipt to $Acct$. Finally, T_2 forwards the request to SP_2. Note that T_2 cannot control what SP_2 would actually do with the forwarded request "*pays(tkID)*", but T_2 can **confidently** confirm that the ticket tkID has been sold to c. The receipt $paid(tID(tkID), cl(c), tP(p), T(t1))$ kept by both T_2 and $Acct$ will support c if any dispute arises.

Rule $\mathcal{R}_2$ of $\overline{\mathcal{L}_2}$ refines the goal **sentCont** delegated from $\mathcal{L}_g$'s *sent* event. If this is the response for a "reserves" request, T_2 extracts SP_2's decision *ans* which is kept in the field *ReservesResult* of *respcnt*. If SP_2 agrees, based on the term $pr(req(reqID), tID(tkID), cl(c), tP(p)$ in CS, T_2 records c's reservation $rs(tID(tkID), cl(c), tP(p), T(t))$. Finally, T_2 forwards SP_2's response to c.

6.3 Law $\mathcal{L}_3$: SP_3's law

Preamble:
law $\mathcal{L}_3$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON arrivedCont([reqID, s, reqcnt]) DO
        cert = SearchST("Cert", reqcnt)
        IF (cert[issuer(CA), Subject(c)]) DO
        {
          IF (nick[cl(c), ni(n)])@CS) DO
          {
             reqcnt1=replaceST(n,"Cert",reqcnt)
             forwardRequest(reqcnt1)
          }
          ELSE DO
          {
             n = generate_nick()
             add(nick(cl(c), ni(n)))
             reqcnt1=replaceST(n,"Cert",reqcnt)
             forwardRequest(reqcnt1)
          }
        }
        ELSE DO
         deliverResponse("invalid certificate")
```

$\mathcal{R}2$.
```
UPON sentCont([reqID, respcnt]) DO
        deliverResponse(respcnt)
```

Figure 6.3: Refinement $\overline{\mathcal{L}_3}$

The main idea is to generate a unique nickname for each client and keep a pair(client, nick) in CS. Each time T_3 receives a request, it checks whether there is a nickname $nick$ associated

with $client$ by looking for the pair(client, nick) in CS. If this is the case, T_3 replaces the client's certificate in the SOAP request by $nick$ and forwards the *modified* request to SP_3; otherwise, T_3 generates a unique nickname $nick$ for the client, adds term "(client, nick)" into CS, replaces the client's certificate (in the SOAP request) by $nick$, and forwards the modified request to SP_3. This way, even though SP_3 does not receive the client's certificate, SP_3 is still able to determine if the request is from a returned client based on the client's nickname. In $\mathcal{R}1$ of $\overline{\mathcal{L}_3}$, we uses the function "findNick" to deal with the searching/creating a nickname for a given client.

$\overline{\mathcal{L}_3}$ is shown in figure 5. $\overline{\mathcal{L}_3}$'s preamble states that $\overline{\mathcal{L}_3}$ is a law refinement of $\mathcal{L}_g$. As in $\overline{\mathcal{L}_2}$, $\mathcal{R}_1$ and $\mathcal{R}_2$ of $\overline{\mathcal{L}_3}$ refine the goals *arrivedCont* and *sentCont* of $\mathcal{L}_g$, respectively.

In $\mathcal{R}_1$, T_3 first verifies the certificate of client c. If it is invalid, then T_3 returns the message "invalid certificate" to c. Otherwise, T_3 checks whether there is a nickname n associated with c. If this is the case, T_3 replaces c's certificate by n and forwards the *modified* request to SP_3; otherwise, T_3 generates a unique nickname n for c, adds term "nick(cl(c),ni(n))" into CS_3, replaces c's certificate (in the SOAP request) by n, and forwards the modified request to SP_3. This way, even though SP_3 does not get c's certificate, SP_3 is still able to determine if the request is from a returned client based on the client's nickname. In $\mathcal{R}_2$, T_3 simply forwards the server's response to c.

6.4 Related work

6.4.1 Trust based on Reputation

The trust issue has been actively investigated for a long time. There are a number of approaches that are built upon reputation for applications in e-commerce, Peer-to-Peer networks (P2P), and SOA. In e-commerce environment, eBay [24] employs one of the earliest reputation system in practice. Ebay buyers can rate seller's service quality after making purchases. The rating can be "positive", "neutral" or "negative". The seller's reputation is the value $P - N$, where P and N are the number of positive and negative ratings, respectively.

In P2P networks, each peer can concurrently be a client or a server which would provide some files to the community. Examples of reputation systems in P2p are [26, 27, 28, 29]. In

[26], Marti et al. proposed a mechanism to collect feedbacks from other peers on a given peer. The reputation of the peer reflects the responses from responding peers and the experience of the requesting peer with the given peer. In [27, 28] a peer is *binary* rated (i.e. a feedback is either negative or positive), while in [29] a peer is judged based on various criteria such as the quality of products the peer sends, the expected delivery time, etc.

Several examples of reputation system for SOA are [30, 32, 33]. In [30, 32] reputation are calculated based on client feedbacks about quality of service. The server reputation are published to a central QoS registry, and made available to potential clients. In [33] Griffiths addresses the decay of trust values over time by proposing a trust decay function. This function takes into account the fact that the experience contributing to the trust values may no longer be relevant. Consequently, the trust values may be outdated and need to be updated.

Approaches based solely on reputation cannot assure the trustworthiness of SOA-based system. In particular, server reputation which is calculated from client feedbacks is not always a good indicator of server trustworthiness because clients may be biased, thus giving inaccurate feedbacks. Moreover, even though reputation can be objectively measured at the client side, say by say installing tamper-resistant devices at client side to measure service quality, having good reputation does not guarantee servers always abide by predefined policies, because servers may sometimes misbehave to gain more benefits.

6.4.2 Service Level Agreement (SLA)

There are many approaches that have been proposed to verify whether service implementation conforms to its service level agreement. They vary in the methods to do the probe (i.e. intrusive or non intrusive), the application phase (test, run-time or posteriori), and the range of verified properties.

In two projects [34, 35], the verification is done by inserting assertions into the code of the server program. Although both projects employ a Trusted Platform Module architecture, clients interacting with a server cannot be really sure that the correct verification has been done, because this verification is very specific to the code of the server. Under SOA, servers may be distributed across multiple administrative domains; and their code of servers may be heterogeneous, may not be available or known to the managers. Therefore such approaches are

not appropriate to provide trustworthiness for SOA-based systems. In our mechanism, policies are explicitly stated in laws providing a clean separation between policies and server programs, and are enforced by LGI controllers out of the control of servers.

There are several research projects which address the testing phase of services, such as [14, 15]. In these works services are assumed to be honest, but their implementation may not be correct. These frameworks provide useful tools for web service developers. Such tools are also useful for clients who want to verify the correctness of the service before using the services. However, these works do not address one of our main concerns: would servers behave as promised?. Note that passing the test phase does not mean a server would always operate as it committed.

Other projects focus on verifying the service behavioral contracts which are defined by the visible interface of services, e.g., [11, 12, 13]. However, they only focus on validating *functional* requirements of web services, and non-functional requirements like server laws presented in this chapter are not addressed.

Several projects only focus on monitoring services, e.g., [36, 37, 38]. In [36], interceptors calculates such properties of services as response time, availability and update those properties to a central registry; clients can inquire the registry to know about server real time performance. In [37], interceptors are responsible for recording certain events, say a server returns a response at 8PM, regarding the client-server interaction in order to provide undeniable of server misbehavior. The project [38] attempts to provide a complete framework for monitoring individual services. In particular, the authors propose both intrusive and non-intrusive probing to monitor 5 system events: client request, service response, application (internal events and states of the service), resource consumption, management (i.e. services reconfiguration, component updating, resource adjusting, message blocking, etc). However, they mainly focus on monitoring functional requirements.

Chapter 7

Orchestration

Under orchestration scheme, a server, called orchestrator, orchestrates several servers to satisfy requests received from its clients. All interactions in the orchestration are conducted via the orchestrator. That means, the orchestrator can intercept, analyze and control flow of messages exchanged between clients and orchestrated servers. As a result, clients and orchestrated servers may not know each other even though clients use services provided by servers. Such *indirect interaction* may cause concerns from both clients and orchestrated servers.

There are two scenarios for an orchestrator O to send a request to a server, as shown in figure 7.1. In the first scenario, O plays the role of an orchestrator and SP_i is one of its orchestrated servers. In the second scenario, O acts as a typical client of SP_k and SP_k does not know that it receives a request from an orchestrator.

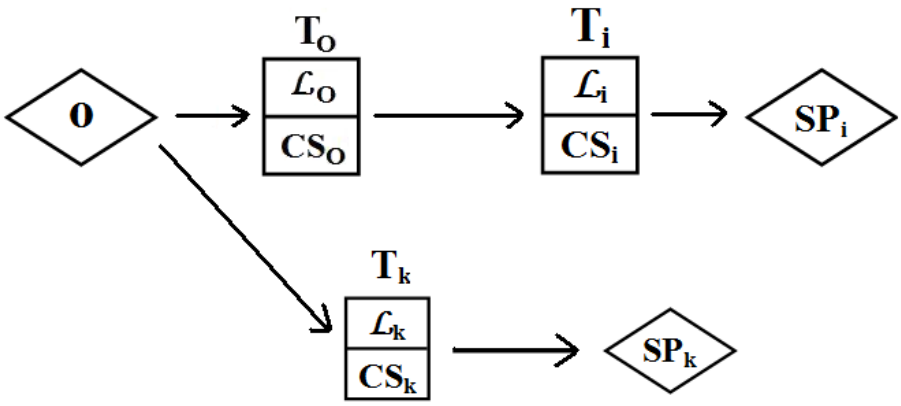

Figure 7.1: Two ways for an orchestrator to invoke a service

If O acts as an orchestrator, the interaction between O and an orchestrated server is governed by the controllers of O and the server. For example, interaction between O and SP_i is regulated by T_O and T_i. If O plays the role of a client when interacting with SP_i, their interaction is only regulated by the server law. In this example, the interaction is governed by T_k

according to $\mathcal{L}_k$. The tradeoff between the two scenarios is that typically response time of a request in the first scenario is longer than that of a request in the second scenario. In the second scenario, exchanged messages are mediated according to only orchestration law $\mathcal{L}_k$. To provide flexibility, our mechanism supports both these scenarios. The second scenario is addressed in chapter 6, hence we would not discuss it further.

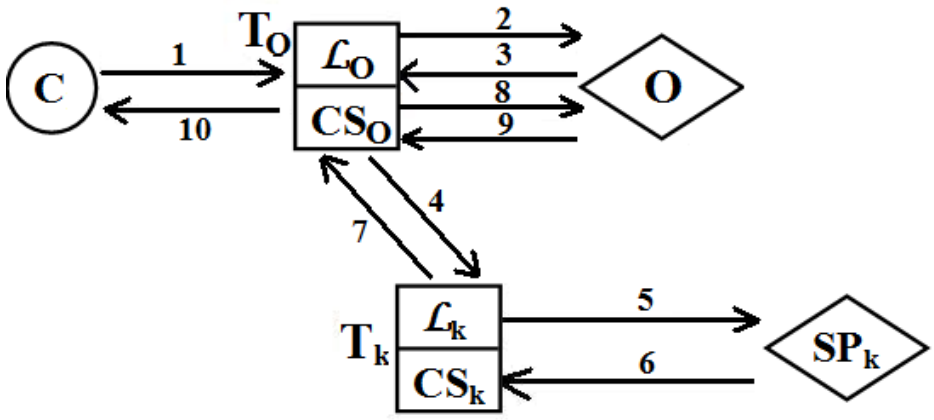

Figure 7.2: Interaction under orchestration scheme

Figure 7.2 shows interaction under orchestration scheme. In step 1, a client C sends a request to invoke a composite service provided by an orchestrator O. The controller T_0 forwards the request to O in step 2. O processes the request and knows that it needs to invoke a service of SP_k in order to satisfy the client request. Therefore, in step 3 O sends a request to T_O which then forwards it to T_k in step 4. Next, T_k forwards the request to SP_k. In step 6, SP_k processes the request, and returns a response to T_k which forwards the response back to T_O in step 7. In step 8, T_O forwards the response to 0. In step 9, O processes SP_k's response and returns the final response to T_O. Finally, T_O delivers the response to C in step 10.

Why does an orchestrator voluntarily send requests via its controller?

In our mechanism, an orchestrator is free to decide how its requests should be treated. Why does O decide to send request as in the first scenario given that it would take longer for its request to reach the orchestrated server and for a response to get back to O? Despite the freedom in deciding how to request a service, an orchestrator may often be forced to send a service request via its controller, and thus subject to its orchestration law $\mathcal{L}_O$, if it wants to receive benefit provided under $\mathcal{L}_O$.

We will present below a list of orchestrator commitments that can be strictly enforced by our mechanism. We then show why orchestrators are effectively compelled to send requests via their own controllers when invoking services.

7.1 Motivating Example

Consider a scenario in which a "travel agent", called an orchestrator under orchestration scheme, orchestrates several "airline" servers to buy a set of tickets for clients. The orchestrator may announce a policy, referred as orchestration policy P_o, to provide certain assurances to clients and orchestrated servers, as follows.

Client concern

Orchestrator acts on client behalf:

When an orchestrator receives a request from a client, it can be considered as the client representative, and starts acting on the client's behalf. Therefore, the client would want the orchestrator not to abuse that privilege. Below is an example to illustrate the need of this assurance.

D1. The client gives its credit card to the travel agent, which will invoke services provided by the airlines to satisfy the client request. The travel agent promises that it only uses the card to pay the airlines, and the total money paid cannot exceed the amount of money the client agrees to pay. Moreover, the travel agent commits that it will not release the credit card data to the airlines, and will delete it right after the client request has been served.

Orchestrator actually completed certain important things for client:

Typically a client has no knowledge about how/when an orchestrator interact with its orchestrated servers to serve the client request. It is hard or even impossible for clients to determine whether an action has been completed by the orchestrator or not. Even if later the client can find out the misbehavior of the orchestrator, but the consequence has been made. Therefore, the client would want to make sure that the orchestrator has actually completed certain things for the client, as stated by the orchestrator.

An example of this assurance is:

D2. The client needs assurance that the travel agent has checked with 3 airlines, and chosen the lowest price available before purchasing a ticket for him/her.

Selection of orchestrated servers:

The execution of a composite service depends on those of orchestrated servers. It may be possible that a client of a composite service expects the orchestrator only use service that comply with certain predefined requirements. Indeed, this expectation is just similar to that coming from clients in real life. For example, when a client buy a car, he/she would want that the car company only utilizes safe and standard-compliance components provided by certified providers.

Below is an example of this assurance.

D3. The travel agent commits to purchase tickets only from airline services that are registered in the directory.

Report truthfully:

Clients may also need to know certain information regarding the actual interaction between the orchestrator and orchestrated servers. Below is an example illustrating the need as well as the assurance made by the orchestrator.

D4. Assume the travel agent purchases airline tickets for the client and receives commission based on the effort it spends. In particular, travel agent receives one dollar for any time it checks with an airline to satisfy the client request. Therefore, the travel agent commits to inform the client about the *actual* number of inquiries it has done on the client behalf.

Orchestrated server's concern

Fairness among orchestrated servers:

Servers may choose to consider orchestrators as a special type of clients, and both the orchestrators and servers can receive benefits from that relationship. For example, servers can reach more clients with the help of an orchestrator, while the orchestrator receives commission for each successful transaction between clients and servers. There is possibly a competition

between *equal servers* in the sense that they provide same services as well as benefit to the orchestrator. How an orchestrator determines *equal servers* depends on business goal of the orchestrator, and is out of the scope of this paper. If this is the case, they would want the orchestrator to not favor some servers, and ignore other servers.

Below is an example of this assurance.

D5. The travel agent promises to treat *equal* airline servers in a fair manner. Specifically, they are placed in a ring, and the travel agent will move along the ring and pick one airline at a time.

Cooperation from orchestrators:

Orchestrated servers do not directly interact with clients, and have to rely on data forwarded from orchestrators to process a request. Therefore, they need the cooperation of orchestrators in providing correct and necessary information about the clients. If full cooperation is met, the orchestrated servers would not waste resource to process invalid requests.

An example of this assurance is shown below.

D6. A client request is defined as *valid* if the client has sufficient funds in his/her credit card account to pay for the ticket(s). The travel agent commits to forward only *valid* requests to airlines.

7.2 Case Study

In this section, we will present the ideas behind the enforcement of orchestration commitments, as well as the law texts. For each orchestration commitment, we will also show why orchestrator is compelled to send requests via its controller.

All rules in the law texts below are refinements of either goal ”sendCont” or goal ”arrivedCont” defined in the global law $\mathcal{L}_g$.

7.2.1 Commitment D1

The client gives its credit card to the travel agent, which will invoke services provided by the airlines to satisfy the client request. The travel agent promises that it only uses the card to pay

the airlines, and the total money paid cannot exceed the amount of money the client agreed to pay. Moreover, the travel agent commits that it will not release the credit card data to the airlines, and will delete it right after the client request has been served.

When O needs to purchase a ticket for a client, it has to send a request via its controller because the orchestrator's controller T_O is keeping credit card of clients.

The idea to implement the constraint (D1) is having the controller of the orchestrator maintain the client's credit card, the amount of money the client agrees to pay, and the amount that has been paid. When the orchestrator needs to use the client's credit card, it needs to send a request via its controller because the controller is keeping the credit card. Upon receiving a request to pay an airline on behalf of the client, the controller makes sure that the amount paid plus the amount being requested do not exceed the amount that the client agreed to pay before letting the request go through. The orchestrator's controller also removes the credit card information from its state when the orchestrator returns a final response to the client about the transaction.

Preamble:
law $\mathcal{L}_{O1}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON sentCont(-,tkPurchase(clientID, tkID, amnt),-) DO
     -- retrieve client's credit card information --
    card(ccInfo, clientID)
     -- retrieve money paid on client behalf, and the amount of
    money agreed --
    finance(amntPaid, amntAgreed, clientID)
    IF (amntPaid + mnt <= amntAgreed)
      {
          newAmntPaid = amntPaid + mnt
          remove(finance(amntPaid, amntAgreed, clientID))
          add(finance(newAmntPaid, amntAgreed, clientID))
         forwardRequest(-, tkPurchase[clientID, tkID, amnt], -)
      }
    ELSE DO
    - inform travel agent, and drop request
     forwardResponse("Money requested exceeds money agreed!")
```

$\mathcal{R}2$.
```
UPON sentCont(-,tkConfirmation(clientID, -),-) DO
    remove(card(-, clientID))
    deliverResponse(-, tkPurchase(clientID), -)
```

Figure 7.3: Refinement $\overline{\mathcal{L}_{O1}}$

In rule $\mathcal{R}_1$, T_O determines this is a purchase request based on the header of the request

tkPurchase. It then retrieves the credit card data for for client whose identity is *clientID*. Moreover, the amount of money paid *amntPaid* using the client's credit card as well as the amount of money the client agrees to pay *amntAgreed* are also retrieved from the control state. If the ticket value *amnt* plus the *amntPaid* do not exceed *amntAgreed*, T_O updates *amntPaid* and forwards the orchestrator's request to the controller of the orchestrated server. As shown, O must send request via its controller in order to use the client's credit card.

In rule $\mathcal{R}_2$, based on the header of the request *tkConfirmation*, T_O knows this is the confirmation of a purchase sent by O to the client *clientID*. The orchestrator is not allowed to use the client credit card any more, thus T_O removes the client credit card. Finally, T_O delivers the confirmation to the client.

7.2.2 Commitment D2

The travel agent commits that it has checked with 3 airline and chosen the lowest price before purchasing a ticket for a client.

In order to use the client credit card, the orchestrator must send *purchase request* to airlines via its controller T_O. However, T_O will not let a purchase request get through if (1) previously O has not sent three inquiries and (2) the amount of money in the *purchase request* is the lowest among three prices returned in three inquiries. This constraint, in turn, forces O to send inquiries via T_O. In summary, the orchestrator is effectively compelled to send request through its controller.

In rule $\mathcal{R}_1$, T_O determines if this is the response of an inquiry based on the message header *tkInquireResponse*. If this is the case, T_O records the number of inquiries O has sent as well as the lowest price of the desired ticket.

In rule $\mathcal{R}_2$, T_O makes sure that the purchase request matches with the responses of three inquiries before forwarding it to corresponding airline.

7.2.3 Commitment D3

The travel agent commits to purchase tickets only from airline services that are registered in the directory.

Preamble:
law $\mathcal{L}_{O2}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON arrivedCont(-,tkInquireResponse(clientID, price, itinary),-) DO
          -- this is a response for an inquiry, increase inquiry number--
          inc(inquiryNumber)
          -- retrieve the lowest price from CS --
          lowestPrice(min, clientID)
          IF (min > price)
            {
                -- update the lowest price if necessary --
                remove(lowestPrice(min, clientID))
                add(lowestPrice(min, clientID))
            }
          -- forward the response of an inquiry to travel agent --
          forwardResponse(tkInquireResponse(clientID, price, itinary))
```

Figure 7.4: Refinement $\overline{\mathcal{L}_{O2}}$

$\mathcal{R}2$.
```
UPON sentCont(-,tkPurchase(clientID, tkID, amnt),-) DO
          -- retrieve the number of inquiries, lowest price from CS --
          inquiryNumber(num)
          lowestPrice(min, clientID)
          IF (num < 3)
            {
             -- number of inquiries is less than 3 --
             -- drop request, tell travel agent about its violation --
             forwardResponse("Need 3 inquiries before making a purchase")
             return;
            }
          IF (min > amnt)
            {
             -- travel agent pays more than recorded lowest price --
             -- drop request, tell travel agent about its violation --
             forwardResponse("Client only agreed to pay lowest price")
             return;
            }
          -- forward valid purchase request to airline --
          forwardRequest(tkPurchase(clientID, tkID, amnt))
```

Figure 7.5: Refinement $\overline{\mathcal{L}_{O2}}$ (continued)

Similar to the cases of commitments (D1) and (D3), orchestrator needs to send purchase request to its controller for approval because the controller maintains the client credit card.

To implement this constraint, the orchestrator's controller needs to get a list of server registered in the service registry. Recall that the service registry is maintaining data about servers that are participate in the system. When the controller receives a request to use a service provided by a server, it will check to see whether the server is in the list. If it is the case, the controller lets the request go through.

It is possible that the list of registered servers the controller maintained is not updated. That happens because servers can leave or join the system in an unpredictable manner, thus only the service registry has the most updated list. A possible approach is to have the registry broadcast the list to orchestrator's controllers when there is a change. However, it may face additional issues. For example, a server joins and then leaves the system; for some reasons, the 'leave' message sent by the registry to a controller is lost; thus, the controller consider the server to be in the system. Alternatively, if a controller does not see a server in its list, it needs to check the server with the registry. Similarly, this would bring about several issues such as either the controller's inquiry or the registry's response is lost. It is quite necessary to consider synchronization issues when applying to real systems. However, for the purpose of demonstration, we do not consider them when implementing this constraint.

Preamble:
law $\mathcal{L}_{O3}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON sentCont(-,tkPurchase(clientID, tkID, amnt),addr) DO
        -- retrieve regList stored in CS --
      regList(list)
      IF (addr in list) DO
           forwardRequest(-, tkPurchase(clientID, tkID, amnt),addr)
      IF !(addr in list) DO
           deliverResponse("Server is not registered in the
      directory")
```

Figure 7.6: Refinement $\overline{\mathcal{L}_{O3}}$

Rule $\mathcal{R}_1$ checks if the destination *addr* of a request is in the register list. The controller T_O retrieves the list of registered servers in the control state. If *addr* belongs to that list, the controller T_O will forward the request further. Otherwise, it will drop the request and inform

the orchestrator.

7.2.4 Commitment D4

Assume the travel agent purchases airline tickets for the client and receives commission based on the effort it spends. In particular, travel agent receives commission, say one dollar, for any time it inquires an airline to satisfy the client request. Therefore, the travel agent commits to inform the client about the ***actual*** *number of inquiries it has done on the client behalf.*

The idea to implement this constraint is to have the controller record all inquiries the controller makes to satisfy a client request. When the orchestrator return a final response to the client, the controller attaches the number of inquiries made.

The orchestrator must sends inquiries via its controller T_O so that it can receive commission at the end of the transaction, so that T_O can calculate the exact the number of inquiries made. Note that an orchestrator is also a server, and it directly interacts with client via its controller, as presented in chapter 6. In other words, the inquiry report sent by the orchestrator to client will be intercepted by T_O. Therefore, T_O is able attach the correct number of inquiry into that report. In sum, the orchestrator is forced to send inquiry as well as inquiry report via its controller in order to get paid.

Preamble:
law $\mathcal{L}_{O4}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON arrivedCont(-,tkInquireResposne(clientID, price, itinerary),-) DO
        -- this is a response for an inquiry, increase inquiry number--
        inc(inquiryNumber)
        add(inquiry(inquiryNumber, price, itinerary))
```

$\mathcal{R}2$.
```
UPON sentCont(-,inquiryReport(clientID,-),-) DO
        inquiryNumber(num)
        report = ""
        (for i=0; i<num; i++)
           strcat(report, inquiry(i, price, itinerary))
        deliver(report)
```

Figure 7.7: Refinement $\overline{\mathcal{L}_{O4}}$

7.2.5 Commitment D5

The travel agent promises to treat equal *airline servers in a fair manner. Specifically, they are placed in a ring, and the travel agent will move along the ring and pick one airline at a time.*

Servers in a ring are assumed to provide equivalent services to serve a client request. We would not discuss further how to determine equivalent services because it depends on business scenarios. For the sake of simplicity, we assume that a manager provides the orchestrator's controller a list of equivalent ones.

Preamble:
law $\mathcal{L}_{O5}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$. **U**PON sentCont(-,tkPurchase(clientID, tkID, amnt),-) DO
-- retrieve ringList stored in CS --
ringList(list)
-- retrieve position of the last airline picked --
curPosition(cur)
n=list.length()
pick = (cur + 1) **a**ddr = list(pick)
-- update current position of ringList in CS --
remove(curPosition(cur))
add(curPosition(pick))
-- send purchase request to the chosen server --
forwardRequest(tkPurchase(clientID, tkID, amnt), addr)

Figure 7.8: Refinement $\overline{\mathcal{L}_{O5}}$

To implement this constraint, the controller places all airlines into a ring, picks one airline to serve a client request, and moves to the next position.

Orchestrator is also forced to send request via its controller. That happens because the orchestrator needs to prove to orchestrated servers that they are treated fairly by sending a service request to T_O, and having T_O fairly choose a server from the ring. Since the service request is forwarded by T_O, orchestrated servers are assured that T_O has fairly picked a server from the ring.

In rule $\mathcal{R}_1$, T_O retrieves the *ring* as well as the position of last airline from the control state. Based on these, T_O picks the airline for this service invocation, updates control state. Finally, it forwards request to the chosen airline.

7.2.6 Commitment D6

A client request is defined as valid *if the client has sufficient fund in his/her credit card account to pay for the ticket(s). The travel agent commits to forward only* valid *requests to airlines.*

Similar to commitment (D5), orchestrated servers can be sure that a client has money to pay for a ticket only if the credit card has been checked by T_O. Therefore, the orchestrator must also send request via T_O.

Preamble:
law $\mathcal{L}_{O6}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON sentCont(-,tkPurchase(clientID, tkID, amnt),-) DO
      -- check whether there is client's card --
    IF (card(ccInfo, clientID) AND sufficient(ccInfo, amnt)) DO
       forwardRequest(tkPurchase(clientID, tkID, amnt))
    ELSE DO
         deliverResponse("credit card problem")
```

Figure 7.9: Refinement $\overline{\mathcal{L}_{O6}}$

In rule $\mathcal{R}_1$, T_O checks if the client has a credit card. It then verifies if the available money in the credit account is greater than the ticket price *amnt* by calling sufficient(ccInfo, amnt). For demonstration purpose, we create a file for storing fictional credit card data. The function *sufficient* simply checks the file. If the available fund is greater than *amnt*, it returns 'true'; otherwise, return 'false'. T_O only forwards the request if the client has sufficient fund in his/her credit card account to pay for the ticket.

7.3 Related Work

All projects discussed in related work section of chapter 6 do not differentiate composite services with individual services. Treating an orchestrator like a normal client brings about flexibility because a servers do not need to distinguish them. Note that our mechanism does *not* exclude this possibility. Not to mention about the quite limited range of client concerns addressed in those projects compared to those addressed by our mechanism as discussed in chapter 6 (e.g., commitments D1, D2, D3, and D4), we believe they are missing one important part when addressing client concerns: the interaction between orchestrators and orchestrated servers.

In the context of building trustworthy service composition, we are not aware of any project

which considers concerns of orchestrated servers. When the orchestrated servers choose to provide some of their services to an orchestrators in order to receive certain benefits as committed by the orchestrators, they need assurance that the orchestrators will fulfill their promises. And, our mechanism provide this assurance by enforcing the orchestrators' commitments.

Chapter 8

Choreography

While the technologies for implementing and interconnecting basic services under SOA are reaching a good level of maturity, modeling choreographies is still a big challenge. The problem is that, current choreography model specify choreographies by focusing on procedural aspects, leading to over-constrained solutions. Moreover, a choreography description in those models is not executable, i.e., there is no enforcement mechanism. Since a single misbehavior of an individual service may destroy the collaboration in a choreography, enforcement is a must to ensure safe and correct coordination between interacting services.

To overcome these limits, we propose that choreography models should focus on specifying the constraints required to make all services collaborate correctly, without stating how such a collaboration is concretely carried out. We believe that services should have the freedom to operate as well as they can, but they must satisfy the choreography constraints through enforcement. Thus, we also present how our mechanism enforces choreography constraints.

8.1 Problems of current choreography model: Procedural approach, no enforcement mechanism

The leading specifications for modeling service choreography e.g., WS-CDL [45], are not capable to to model choreographies, thus failing to solve their objective. The main problem is that they models choreographies by focusing on procedural aspects, e.g. by specifying control and message flow of the interacting services. This forces the choreography modeler to capture it at a procedural level. It is important to note that the implicit assumption behind procedural models is that *anything that is not explicitly defined is not allowed to occur*. Therefore, a choreography modeler must list all allowed executions when using those choreography specifications.

We have discussed the deficiencies of procedural approach in [42]. One apt criticism of the

approach is pointed out in [43], ”Business processes are highly dynamic and unpredictable–it is difficult to give a complete a priori specification of all the activities that need to be performed and how they should be ordered. Any detailed time plans which are produced are often disrupted by unavoidable delays or unanticipated events.”

As an alternative for procedural specification, these authors propose the following: ”the most natural way to view the business process is as a collection of autonomous problem solving agents, which interact when they have interdependencies.”

We too believe in the importance of autonomy for the participants in choreographies, for the above mentioned reason, and for others. In particular, a participant can take ”opportunity-based initiatives” [44], based on his/her intimate familiarity with the operating environment, which may not be available to the manager. An attempt to communicate such information to some manager(s) could be impractical when fast response is required, particularly in a distributed setup, when the agent in question is geographically separated from his manager(s).

We argue that a choreography model should provide both compliance and flexibility: on the one hand, all interacting services must strictly follow the constraints defined in the choreography; on the other hand, each of them should have the autonomy to execute the business processes which cover its own contribution to the choreography. In other words, we believe that a service choreography need to focus on the rules of engagement required to make all the interacting parties collaborate correctly, without stating how such a collaboration is concretely carried out.

Last but not least, there is no enforcement mechanism provided for current choreography. And, it will be impossible to come up with one for enforcing choreography descriptions as they are currently modelled. Instead, currently choreography descriptions can only be used for describing the rules of engagement for making all the interacting services collaborate correctly, and for verification purposes (such as conformance checking and interoperability). It is up to the interacting services to implement the choreography rules, and there is no guarantee that all services abide by the choreography rules because some services may misbehave to get some benefits. We believe that this is unacceptable because even a single violation, either intentionally or unintentionally, would destroy the reliability of the system, thus preventing effective collaboration under choreography scheme.

8.2 Our Approach

To overcome the limits of procedural approaches, we propose that choreography models should focus on specifying the constraints required to make all services collaborate correctly, without stating how such a collaboration is concretely carried out. Our proposed mechanism allows modelers to define choreographies through a set of policies or business rules referred to as constraints, which will be strictly complied by interacting services. In our mechanism, despite the communal nature of choreography constraints, they are locally enforced at each service based on the service's state, and without having any direct knowledge of the coincidental state of other services. This locality enhances the efficiency and scalability of law-enforcement, without reducing the expressive power of the choreography constraints.

As we saw in chapter 6, a service may be associated with a law which would specify constraints about the behavior of the service. That law may take into account the various functions provided by the service, as well as the information about the domain in which the service is being deployed. In other words, that law - referred as server law - addresses specific features regarding one particular service of the system. Note that server law is optional, i.e., not all services have their own laws.

Moreover, in SOA-based systems a single service may concurrently participate in several choreographies, each of which is subject to a choreography law. Note that this service may also be subject to its server law. The behavior of the service in one choreography is subject to both the server law as well as the choreography law imposed on all members of the choreography.

That is just similar to what happens in real life. For example, a Rutgers professor may specify his own law $\mathcal{L}_P$ in which he define his specific expertise and commitments with his students/department. Note that the professor's law conform to his Rutgers university law, which is viewed as a global law. He may choose to join his department admission committee, a labor union, and the committee of a conference in his field. Assume that the laws that regulate the department admission committee, the labor union and the conference committee are called $\mathcal{L}_A$, $\mathcal{L}_L$ and $\mathcal{L}_C$, respectively. In this scenario, the $\mathcal{L}_A$, $\mathcal{L}_L$ and $\mathcal{L}_C$ are considered as choreography laws.

There are two scenarios that need to be addressed.

1. Server law is defined before the choreography law

How can a server law allow its service to join a choreography given that the server law is defined **before** the choreography law? For example, $\mathcal{L}_P$ is defined before $\mathcal{L}_C$. In this scenario, the server law is **not aware** of the choreography law.

The joining time:

At the server's side, we note that even though server law *does not know any* ***specific*** *choreography law defined after it*, it knows that its service is requesting to join a choreography, so it still can control the participation of the service in the choreography. For example, the server law states that its service cannot join more than two choreographies at any moment, while another server law states that its service can join a new choreography if the request is sent between 8AM and 6PM.

At the choreography's side, the law maker of a choreography can control what servers are allowed to participate in the choreography. For example, a labor union of a company may state that only employees of the company can join the union.

The interaction time

If choreography law is aware of server law, it may treat the service differently from others. For example, when the professor joins the labor union, $\mathcal{L}_L$'s law makers may place more confidence on him because they know that the professor's behavior is also subject to Rutgers university law. Even if the text of the Rutgers law is not known, it may be enough to know that this is the official law of Rutgers, and thus worthy of certain trust.

The behavior of a service in a choreography is governed by both the server law and the choreography law. For example, the behavior of the professor in a committee is governed by the professor law and the committee law. In particular, the professor is subject to $\mathcal{L}_P$ and $\mathcal{L}_A$ when he is a member of the admission committee, and is subject to $\mathcal{L}_P$ and $\mathcal{L}_C$ when he is serve the conference committee.

2. Server law is defined after the choreography law

The joining time:

The server law can be aware of the choreography law. Therefore, it can *explicitly* allow or disallow the service to join the choreography. However, it can also choose to treat the choreography as it does not know this specific choreography law.

The interaction time:

Since the choreography law is not aware of the server law, it cannot have a *specific* treatment for this *particular* server law. Instead, it can provide a *standard* treatment for all server laws it does not know. For example, the choreography law state that any agent that is operating under a server law can join the choreography. Finally, the behavior of a service in a choreography is subject to both the server law and the choreography law, as in the previous scenario.

How LG-SOA mechanism supports these two scenarios will be presented in detailed shortly later.

Discussion:

There is **no** requirement about the order in which laws are specified. More specifically for this scenario, the professor's law $\mathcal{L}_P$ is *not* required to be defined *before* the conference committee's law $\mathcal{L}_C$, or *vice versa*. And, the same thing occurs for the case of $\mathcal{L}_P$ and $\mathcal{L}_A$. Otherwise, the flexibility and applicability of the system would be hindered. For example, if the professor's law $\mathcal{L}_P$ is required *before* the conference committee's law $\mathcal{L}_C$, then any professors whose laws are defined *after* $\mathcal{L}_C$ are not allowed to join the conference committee.

8.3 A need to extend LGI

Consider a choreography C involving one unconstrained software actor called Z, two servers X and Y operating under laws $\mathcal{L}_X$ and $\mathcal{L}_Y$ which are subordinate to $\mathcal{L}_g$, thus we have $\mathcal{L}_X$-agent, and $\mathcal{L}_Y$-agent. The actor Z is not governed by a law, so we called it an unconstrained actor.

Let the constraints on C be defined by law $\mathcal{L}_C$. For the simplicity of discussion, we only present two server laws $\mathcal{L}_X$, $\mathcal{L}_Y$ and one choreography law $\mathcal{L}_C$. Our discussion below, however, is applicable to any scenario which consists of a large number of server/choreography laws.

Recall that LGI does not coerce any actor to exchange $\mathcal{L}$-messages, under any specific law $\mathcal{L}$, or to engage in LGI-regulated interaction in any other way. Such an engagement is purely voluntary. LGI can nevertheless be said to enforce its laws in the following sense: if an LGI-agent X interacts with a process Y claimed to be an LGI-agent operating under law $\mathcal{L}_X$, then X can be confident that Y conforms to law $\mathcal{L}$. It is this confidence that allows the members of a given $\mathcal{L}$-community to trust each other. Despite this voluntary engagement in an LGI-regulated

activity, an agent may often be forced to operate under a particular law $\mathcal{L}_X$, if he/she wishes to use services provided only under this law.

In chapters 5 and 6, global law and server law are enforced only if clients send request to server's controller. But, why do clients do that? In fact, clients choose to sent their requests to the controller of the server because of the following reasons:

1. Clients may not know the real address of a server. Recall that only registered services are maintained in the registry. And, the registry provides clients with location of the server's controller.

2. Even if a client knows the real address of the server, the client may still send request to the server's controller because clients want assurance that server commitments are enforced. And, if this is the case, global constraints are also enforced.

3. Besides, servers may *require* clients send requests via their controllers, and would not process requests sent directly from clients. One scenario that shows a server chooses to do so is that it wants to show their usefulness, e.g., serving many requests, to a manager. However, the manager does not trust the servers in reporting correctly the number of client requests and only accepts the report sent by the server's controller; consequently, servers require their clients send requests to their controllers.

If no one has interest in forcing clients to send requests via controllers, we face a common problem on the Internet: we cannot force someone to do something if they do not get benefit by doing this.

In chapter 7, under orchestration scheme an orchestrator can act as a client or an orchestrator when it needs to invoke a service. Even though the orchestrator is free to decide which role to play, in some scenarios it voluntarily chooses to act as an orchestrator by being an orchestrator. The same principle applies here. It is our view that actors participate in a choreography because they would like to receive benefits as a choreography member.

Consider the interactions in the choreography. For ease of discussion, we will discuss interactions that involve X; similar things happen for other choreography members such as Y and Z–a software actor that is not being regulated by a law. The first type of interaction is between X and its clients, and is only regulated by $\mathcal{L}_X$. The second one is between X and

other $\mathcal{L}_C$ choreography members. This type of interaction is subject to $\mathcal{L}_C$, $\mathcal{L}_X$, and the law of the actor which is interacting with X. For example, the interaction between X and Y is subject to $\mathcal{L}_X$, $\mathcal{L}_Y$, and $\mathcal{L}_C$. The interactions as well as laws that govern them are represented in figure 8.1.

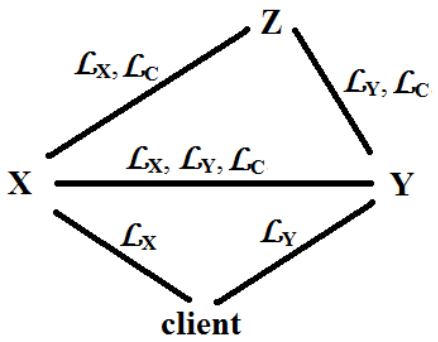

Figure 8.1: Interaction types and Laws that govern them

Take a closer look at X. To enforce laws on two types of interactions of X, we need two type of agents. The first type is regulated *only* by a law–$\mathcal{L}_X$ in this scenario, while the second type is regulated by *both* two laws–$\mathcal{L}_C$ and $\mathcal{L}_X$ in this case.

The standard LGI model does **not** support the second type of agents because it does not allow an LGI-agent to join a LGI community. In this scenario, the $\mathcal{L}_X$-agent cannot participate in $\mathcal{L}_C$-choreography under the standard model. Therefore, we have extended LGI to enable the possibility that the $\mathcal{L}_X$-agent can join $\mathcal{L}_C$-choreography. We will present in detailed the LGI extension in section 8.4.

Now, we will discuss how laws are organized in LGI hierarchy.

We cannot place it above $\mathcal{L}_X$ and $\mathcal{L}_Y$ as in figure 8.2.a because of two reasons.

First, this would make them subordinate to $\mathcal{L}_C$. That means, the interaction between X and its client is governed by both $\mathcal{L}_C$ and $\mathcal{L}_X$, and the interaction between Y and its client is governed by both $\mathcal{L}_C$ and $\mathcal{L}_Y$. That is not what we want because as shown in figure 8.1 the interactions between X and its clients, between Y and its clients are supposed to be governed **only** by $\mathcal{L}_X$ and $\mathcal{L}_Y$, respectively.

Second, it is possible that $\mathcal{L}_X$ (or $\mathcal{L}_Y$) is defined *before* $\mathcal{L}_C$, thus $\mathcal{L}_C$ cannot be a superior

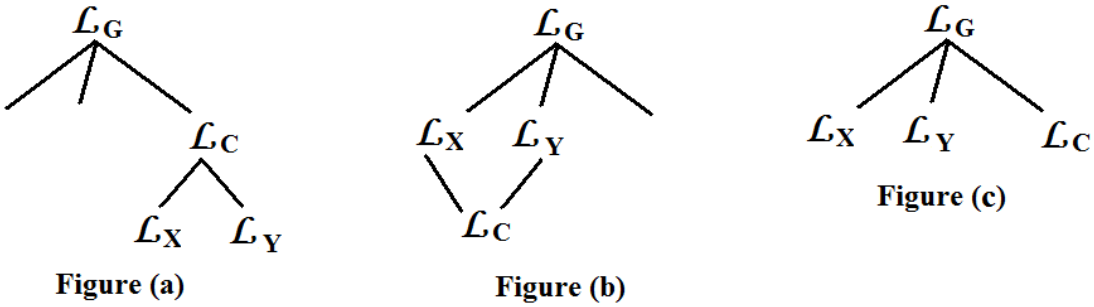

Figure 8.2: Law Organization

of $\mathcal{L}_X$ (or $\mathcal{L}_Y$). Recall that in LGI-hierarchy, a superior of a law must be defined *before* that law. For example, $\mathcal{L}_g$ must be defined before $\mathcal{L}_X$ and $\mathcal{L}_Y$.

Besides, we cannot place $\mathcal{L}_C$ as in figure 8.2.b because a law has *at most one* superior in the law hierarchy.

We choose to place laws as in figure (8.2.c) because there is no order in which server laws and choreography laws are defined.

To enforce choreography constraints we need a mechanism that has the following five properties:

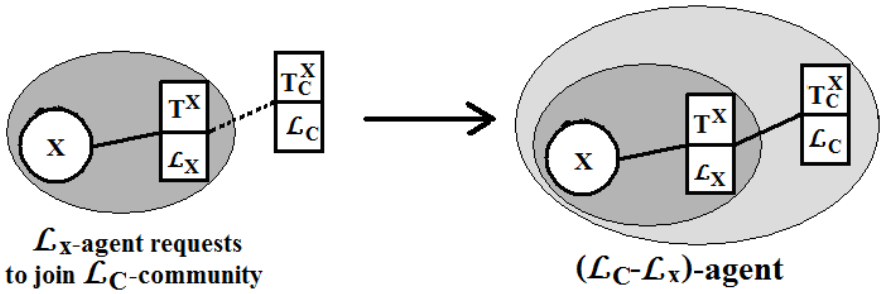

Figure 8.3: ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent

First, it allows a software actor X which is currently governed by $\mathcal{L}_X$ to join a choreography regulated by $\mathcal{L}_C$. That means the mechanism needs to support the possibility that an LGI-agent to be regulated by another law. As shown in the figure below: Actor X operates under $\mathcal{L}_X$, and we have $\mathcal{L}_X$-agent; the $\mathcal{L}_X$-agent joins $\mathcal{L}_C$-choreography. So we have two agents $\mathcal{L}_X$-agent and ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent as shown in figure 8.3. The notation ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent indicates that an $\mathcal{L}_X$-agent adopts a controller to participate in $\mathcal{L}_C$-choreography.

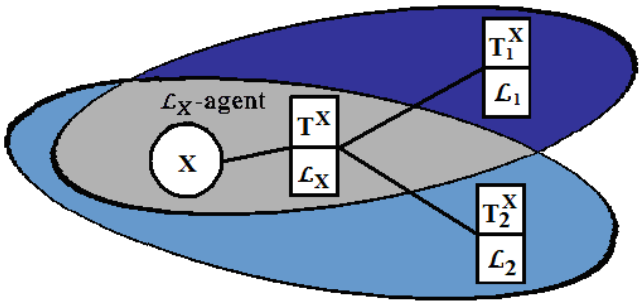

Figure 8.4: $\mathcal{L}_X$-agent join $\mathcal{L}_X$ and $\mathcal{L}_X$ choreographies

Second, it is possible that an $\mathcal{L}_X$-agent joins more than one choreography. For example, figure 8.4 shows that an $\mathcal{L}_X$-agent joins $\mathcal{L}_1$-choreography and $\mathcal{L}_2$-choreography, so we have two newly created agents: ($\mathcal{L}_1$ - $\mathcal{L}_X$)-agent) and ($\mathcal{L}_2$ - $\mathcal{L}_X$)-agent.

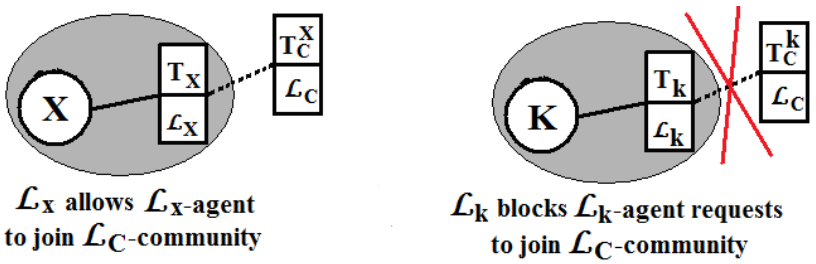

Figure 8.5: A law can allow/disallow its agents to join a choreography

Third, it is possible that a choreography law $\mathcal{L}_C$ is defined after server laws. Therefore, a server law $\mathcal{L}$ may not be aware of any *specific* choreography law. However, our mechanism allows $\mathcal{L}$ to decide whether an $\mathcal{L}$-agent is allowed to join a new choreography $\mathcal{L}_C$ and under what conditions. For example, as shown in figure 8.5, $\mathcal{L}_X$ allows $\mathcal{L}_X$-agent to join *any* choreography if a join request is sent from 8AM-10PM while $\mathcal{L}_k$-agent is not allowed to join a new choreography because it has joined two other choreographies.

Fourth, our mechansim provides the choreography managers the capability to distinguish the type of the actor requesting to join the choreography, and decide to allow the actor to join the choreography or not. For example, the managers can write $\mathcal{L}_C$ in a way which can control what types of actor are allowed to join $\mathcal{L}_C$-choreography. As shown in figure 8.6, all agents except $\mathcal{L}_V$-agent can join $\mathcal{L}_C$-community.

Note that this scenario is different from the one presented in figure 8.5 in term of what law

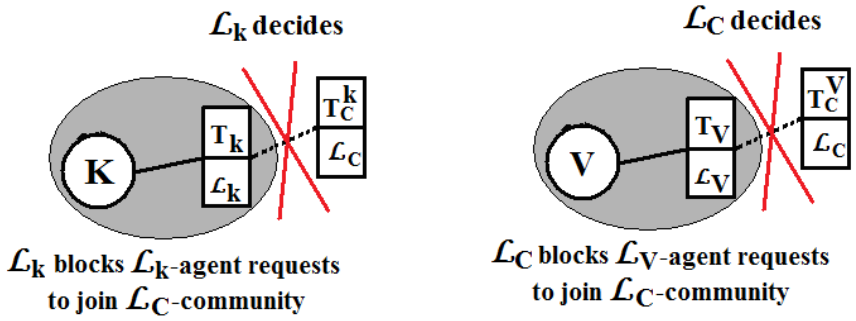

Figure 8.6: 2 cases in which an agent is not allowed to join a choreography

prevents the participation of an actor in a choreography (even though results are the same: $\mathcal{L}_k$-agent cannot join $\mathcal{L}_C$-choreography). Specifically, in figure 8.5 $\mathcal{L}_k$ does not allow $\mathcal{L}_k$-agent to join $\mathcal{L}_C$-choreography, while in figure 8.6 $\mathcal{L}_V$ allows $\mathcal{L}_V$-agent to join new choreographies but $\mathcal{L}_C$ does not.

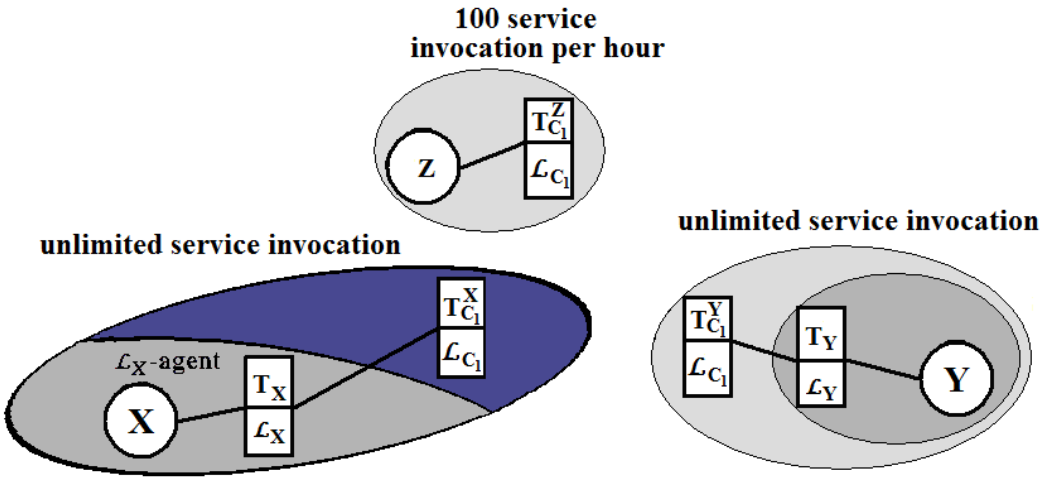

Figure 8.7: Different types of actors are treated differently

Finally, as there may be different types of actors in $\mathcal{L}_C$-choreography, it is possible that $\mathcal{L}_C$ managers choose to treat them differently. As shown in figure 8.7, an unconstrained actor like Z can only invoke 100 services per hour provided by other $\mathcal{L}_C$-choreography members while $\mathcal{L}_X$-agent and $\mathcal{L}_Y$-agent can invoke as many services as they wish.

Discussion:

Several observations about the participation of $\mathcal{L}_X$-agent in $\mathcal{L}_C$-choreography are in order.

- First, the $\mathcal{L}_X$-agent can participate in $\mathcal{L}_C$-choreography only if both laws $\mathcal{L}_X$ and $\mathcal{L}_C$

approve.

- Second, the conditions under which a "join $\mathcal{L}_C$-choreography" request of $\mathcal{L}_X$-agent can be accepted, and the effects of such an action are determined by two laws.

- Third, why would $\mathcal{L}_C$ want to know the types of members that join $\mathcal{L}_C$-choreography? There can be several reasons for this. First of all, familiarity with the law that govern a member's behavior gives one some needed confidence. For example, assume that X is a professor at Rutgers which would have to obey a Rutgers policy for professor; when X joins a labor union $\mathcal{L}_C$ as a Rutgers professor, $\mathcal{L}_C$ managers may place more confidence on X because she knows that X is also subject to Rutgers policy. Second, even if the text of actor law $\mathcal{L}_X$ is not known, it may be enough to know that this is the official law of Rutgers, and thus worthy of certain trust.

8.4 Activities under choreography scheme

In this section, we will present the process for an $\mathcal{L}$-agent to join a new choreography. Then, we describe what laws and events that govern the interaction between any two choreography members.

8.4.1 An LGI-agent joins a choreography

Consider a scenario in which an actor X operates under $\mathcal{L}_X$. Assume that X wants to join $\mathcal{L}_C$-choreography.

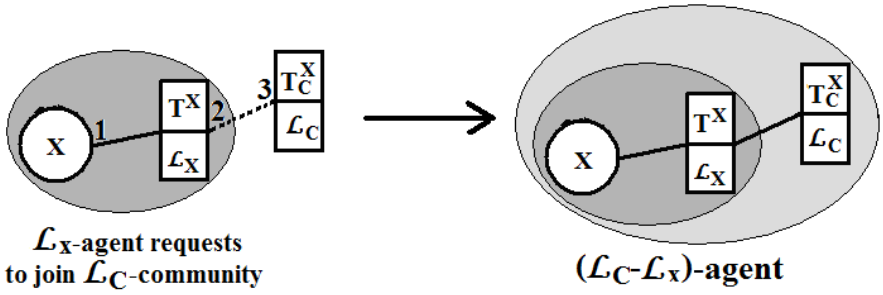

Figure 8.8: newAdopted Event

Figure 8.8 shows the process in which an $\mathcal{L}_X$-agent adopt a controller to become a new member of $\mathcal{L}_C$ choreography.

In step 1, X sends a message of the form *joinCommunity(contHost, commLaw, shortName)* to T^X where:

- *conHost* is the name of the controller that X wants to adopt in order to join the choreography.

- *commLaw* is the law governing the choreography

- *shortName* is the desired short name of X in the *commLaw*-choreography (i.e., the full name of the new agent would be *shortName@conHost*).

For this figure, *conHost* is T_C^X, *commLaw* is $\mathcal{L}_C$, and *shortName* is X.

In step 2, the *joinCommunity* message arrives at T^X and an event *joinRequest(String contHost, String commLaw, String shortName)* is fired. Because the *joinRequest* event is fired, even though $\mathcal{L}_X$ may not be aware of any *specific* choreography law defined after the birth of $\mathcal{L}_X$, it is possible to write $\mathcal{L}_X$ in a way that enable the controller enforcing $\mathcal{L}_X$ to know there is a request to join a choreography from X.

T^X evaluates the *joinRequest* event according to the law $\mathcal{L}_X$. If $\mathcal{L}_X$ does not allow X to join a new choreography, then T^X drops the request. This is the capability we provide a law to control the participation in a new choreography of its agents.

If $\mathcal{L}_X$ allows X to join a new choreography, it forwards the *joinCommunity* message to the controller *conHost*, thus an event *newAdopted(preLaw, preHost, shortName)* is fired at *conHost* in step 3 where:

- *preLaw* is the law of the LGI agent requesting to join the choreography.

- *preHost* is the name of the controller of the LGI agent.

For this figure, *preLaw* is $\mathcal{L}_X$ and *preHost* is T^X.

In step 3, T_C^X evaluates the *newAdopted* event according to *commLaw*. Note that the event newAdopted is defined in the law *commLaw*. If *commLaw* does not allow X to join the choreography, there will be an operation "*doQuit*" in the *newAdopted* event. This operation will not allow the agent *shortName@conHost* to exist. For more information about "*doQuit*" operation, the reader is referred to the LGI manual.

As shown, a choreography law can decide what types of actors can join the choreography and under what condition.

If *commLaw* does allow X to join the choreography, a new agent shortName@conHost is created:

- The hash of *preLaw* is added to the control state of shortName@conHost to indicate that its actor is an LGI agent and governed by *preLaw*.

- *preHost* is also recorded in the control state so that messages sent to *shortName@conHost* will be forwarded to *preHost* for evaluation according to *preLaw* before being delivered to X. How incoming/outgoing messages of this agent are regulated is presented in the following section.

After this process, we have a newly created ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent. The notation ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent means that an $\mathcal{L}_X$-agent adopts another controller to be a member of $\mathcal{L}_C$-choreography.

In summary, LGI-agent can only join a choreography only if *both the law that is governing it and the choreography law allow* the LGI-agent to join the choreography.

8.4.2 Interaction between members of a choreography

There are two types of members in the $\mathcal{L}_C$-choreography:

- normal members are software actors whose behaviors are regulated only by $\mathcal{L}_C$. An example is the unconstrained actor Z shown in figure 8.7.

- LGI-agent members whose behaviors are regulated by $\mathcal{L}_C$ as well as their laws. For example, we can have ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent, and ($\mathcal{L}_C$-$\mathcal{L}_Y$)-agent.

Consider an example $\mathcal{L}_C$-choreography consisting of three members: one ($\mathcal{L}_C$-$\mathcal{L}_X$)-agent, one ($\mathcal{L}_C$-$\mathcal{L}_Y$)-agent, and two normal members Z and V, i.e, Z and V are $\mathcal{L}_C$-agents.

There are many interactions between members in a choreography. However, these interactions can be categorized into four types of message exchanges. We will describe each type and present an example of it for clarification.

1. the sending of a message from an LGI-agent member to another LGI-agent member. For example, X sends a message to Y.

2. Second, the sending of a message from an LGI-agent member to a normal member. For example: X sends message to Z.

3. Third, the sending of a message from a normal member to an LGI-agent member. For

example: Z sends message to Y.

4. Fourth, the sending of a message from a normal member to another normal member. For example: Z sends message to V. This interaction is similar to that in standard LGI mechanism, so we will not discuss it further.

Interaction between two LGI-agent members

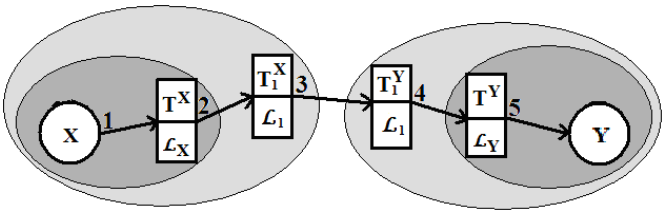

Figure 8.9: ($\mathcal{L}_1$-$\mathcal{L}_X$)-agent sends a message to ($\mathcal{L}_1$-$\mathcal{L}_Y$)-agent

The interaction between $\mathcal{L}_X$-agent member and $\mathcal{L}_Y$-agent one in $\mathcal{L}_1$-choreography is presented in figure 8.9.

In step 1, X sends to T^X a message in the format of *newSendLG(controller List, String message, String destination)* where:

- *destination* is the LGI name of receiver

- *message* is the message the sender wants to send to the receiver

- *controller List* is the set of controllers that the *message* will be passed through before reaching the destination.

For the example in this figure, *controller list* is T^X, T_1^X, and *destination* is Y@T_1^Y.

In step 2, the arrival of the *newSendLG* message at T^X leads to the firing of an event *sendRequest(controller list, String message, String destination).* T^X evaluates the event according to $\mathcal{L}_X$, removes its address (i.e., T^X) from the *controller list* and forwards the *newSendLG* message to T_1^X, which is the next controller in the *controller list*. For this figure, at this step the controller list is T_1^X.

In step 3, when the *newSendLG* message reaches T_1^X, another *sendRequest(controller list, String message, String destination)* event is fired. Similarly, T_1^X evaluates the event according to $\mathcal{L}_1$, and removes T_1^X from the *controller list*. As the result, the *controller list* is empty, and T_1^X knows that it is the *sender's last* controller which is responsible for regulating the

newSendLG message. Thus, it is time for T_1^X to forward message to *destination* running on T_1^Y.

In step 4, when message arrives at T_1^Y, an *newArrived(String source, String message, String destination)* is fired. Based on its control state and the parameter destination, which is Y@T_1^Y, T_1^Y knows that it needs to deliver the message to T^Y.

Finally, in step 5, another *newArrived(String source, String message, String destination)* is fired at T^Y which will deliver message to Y.

LGI-agent members send messages to normal members

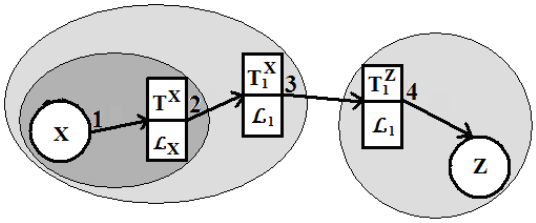

Figure 8.10: ($\mathcal{L}_1$-$\mathcal{L}_X$)-agent sends a message to $\mathcal{L}_1$-agent

Step 1, 2, and 3 in figure 8.10 are similar to those in figure 8.9. In step 4, *newArrived(String source, String message, String destination)* event is fired at T_1^Z which will evaluate the event according to law $\mathcal{L}_1$, and finally deliver the message to Z. This step is similar to step 5 in figurefig-89.

Normal members send messages to LGI-agent member

In step 1 of figure 8.11, Z sends to T_1^Z a message in the format of *newSendLG(controller List, String message, String destination)* where *destination* is the LGI name of the expected receiver of *message*, and *controller List* is the set of controllers that the message will be passed through before reaching the *destination*. In this figure, *controller list* is T_1^Z, and destination is Y@T_1^Y.

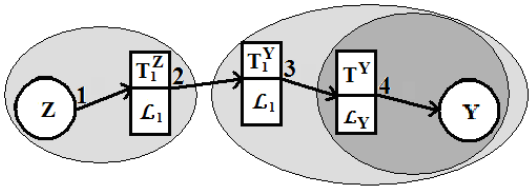

Figure 8.11: $\mathcal{L}_1$-agent sends a message to ($\mathcal{L}_1$-$\mathcal{L}_Y$)-agent

In step 2, when the newSendLG message arrives at T_1^Z, a *sendRequest(controller list, String message, String destination)* event is fired. T_1^Z evaluates the event according to $\mathcal{L}_1$, and removes T_1^Z from the *controller list*. Because the *controller list* is empty now, T_1^Z knows that it is the *sender's last* controller. Therefore, it will forward the message to *destination* running on T_1^Y.

Step 3 and 4 are similar to step 4 and 5 respectively in figure 8.9, so we will not discuss them further.

8.5 Case Study

Consider a company that has multiple departments, each of which consists of many employees. An employee working for a department d is subject to $\mathcal{L}_d$, the department's law. For example, an employee in the Transport and Storage Department is subject to law $\mathcal{L}_{TS}$ while an employee working in the Marketing Department is governed by law $\mathcal{L}_M$.

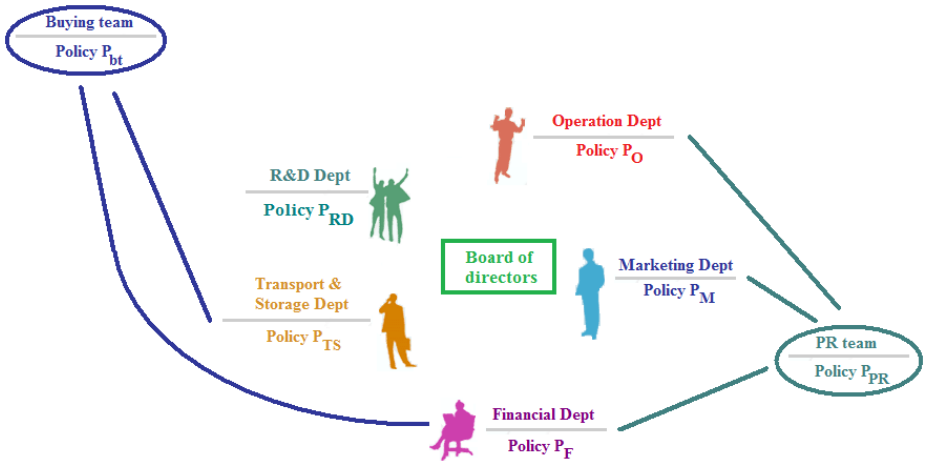

Figure 8.12: Motivating example

Later, the company deploys a team of employees, each of whom is working in a department. In this example, the team members are from Transport Storage Department, and from Financial Department. The responsibility of the team is to supply the company with the merchandise it needs.

The team consists of a manager, and a set of employees authorized as buyers, via a purchasing-budget provided to them. The buyers are supposed to operate autonomously in deciding what to buy, at which price, and from whom-but they are limited by their purchasing budget, which

they can obtain from the manager, or from fellow buyers.

The manager needs to be able to monitor the purchasing activities of the various buyers, and to steer these activities by adjusting their budgets, as it sees fit. Such monitoring and steering are carried out even if the set of active buyers, and the manager itself, change dynamically, during the purchasing activity. The proper operation of this buying team would be ensured if all its members comply with the law called $\mathcal{L}_{BT}$. The buying team have been discussed used in several LGI-related projects. We will presented an adapted version of $\mathcal{L}_{BT}$ later shortly to demonstration the usage of our mechanism.

Similarly, we can have a public relation team whose members are from Operation Department, Marketing Department, and Financial Department. The PR team is regulated by the law $\mathcal{L}_{PR}$. In this example, choreography constraints are be defined after all other policies are deployed. Specifically, $\mathcal{L}_{BT}$ and $\mathcal{L}_{PR}$ are defined after $\mathcal{L}_{TS}$, $\mathcal{L}_{O}$, $\mathcal{L}_{F}$, $\mathcal{L}_{M}$, are defined. Also, an employee may concurrently join several choreographies, as in the case of those in Financial Department.

Because the laws $\mathcal{L}_{TS}$, $\mathcal{L}_{F}$ are defined before $\mathcal{L}_{BT}$, it is possible that the law manager of $\mathcal{L}_{BT}$ is aware of them. It is also possible that the law manager would take that awareness into account when defining $\mathcal{L}_{BT}$. Specifically, the he/she may choose to treat them differently if he/she wants. Below is an adapted version of $\mathcal{L}_{BT}$:

1. The assignment of actors to roles: To hold the role of a manager or of an auditor one needs to be authenticated by a certificate signed by a specified certification authority (CA). And to hold the role of a buyer, one needs to be appointed to it by the manager.

2. A manager can provide each buyer in her team with a purchasing budget, and can later change it at will. The initial budget of a team member from financial department and transport-storage department is 1,000 and 800 currency units, respectively. Team members from other departments only receive an initial budget of 500 currency units.

3. Buyers are allowed to issue purchase orders (POs) for an amount not to exceed their budget which would be decremented by this amount. A copy of each such PO is to be sent to the manager.

4. Buying team members who come from the financial department can make as many POs as they wish, while buying team members who come from the transport-storage department are

only allowed to make at most 10 POs per day. Other team members are limited to three POs per day.

Assume that law of financial department $\mathcal{L}_F$ is simply described as follows:

1. The head of the department is not allowed to participate in any cross-department team.

Also, assume that $\mathcal{L}_{TS}$, the law of transport-storage department, is below:

1. An employee is not allowed to join more than one cross-department team.

For demonstration purpose, we have intentionally simplified the three laws $\mathcal{L}_F$, $\mathcal{L}_{TS}$, and $\mathcal{L}_{BT}$. Moreover, for the sake of simplicity, we do not provide a company law which, if exists, would serve as a global law to assure certain uniformity across the company. However, our mechanism works well if a company law is needed.

Even though $\mathcal{L}_F$, $\mathcal{L}_{TS}$ and $\mathcal{L}_{BT}$ are simple, they still demonstrate several important points:

First, the department laws ($\mathcal{L}_F$, $\mathcal{L}_{TS}$) and the team law (or choreography law $\mathcal{L}_{BT}$) have the capability to control the participation of an employee in the team. And, an employee can join a team only if he/she has been approved by *both* the department law and the team law.

Second, the team law (or choreography law) is able to distinguish team members and to treat them differently if necessary.

Third, the choreography law can be defined after the department law as shown. Another interesting thing is that $\mathcal{L}_{BT}$) can accommodate team members from a newly established department (i.e., the law of the new department law is written *after* the birth of $\mathcal{L}_{BT}$) without the need to made any change because it can provide a ***standard treatment*** for members who are not from financial or transport-storage departments. Specifically, the constraints "Team members from other departments only receive an initial budget of 500 currency units", "other team members are limited to one PO per day" are example of ***standard treatment***.

8.5.1 Law $\mathcal{L}_F$

In rule $\mathcal{R}_1$, the employee's certificate is verified. Based on the attribute *att* of the certificate, a corresponding role is added to control state.

In rule $\mathcal{R}_2$, the controller retrieves from the control state the role the employee plays. If the employee's role is head of the department, then the controller drops the *joinCommunity* request

Preamble:
law $\mathcal{L}_F$ refines $\mathcal{L}_g$.

$\mathcal{R}1$. **U**PON adopted(-, cert[issuer, subject, att]) DO
-- verifying certificate, skipped ---
-- add employee's role to control state ---
-- role is either "employee" or "Head" ---
add(role(att))

$\mathcal{R}2$. **U**PON joinRequest(conHost,commLaw, shortName) DO
-- retrieve role of this employee --
role(r)
IF (r != "Head") DO
forward(joinCommunity(-,commLaw, shortName),conHost)
ELSE DO
deliver("Head of department cannot join any team)

Figure 8.13: $\mathcal{L}_F$ Financial Department Law

and informs the employee that he/she is not allowed to join any choreography. Otherwise, the controller forward the request to *conHost*, which would be the controller enforcing *commLaw* when the employee joins *commLaw*-choreography.

8.5.2 Law $\mathcal{L}_{TS}$

In rule $\mathcal{R}_1$, the employee's certificate is verified. The the team number value is initialized with zero, indicating that the employee does not participate in any team.

In rule $\mathcal{R}_2$, the controller retrieves team number value from the control state. If that value equals zero, the employee is allowed to join the team: the team number value is increased by one, and the *joinCommunity* request is forwarded to *conHost*. Otherwise, the controller drops the *joinCommunity* request and informs the employee.

8.5.3 Law $\mathcal{L}_{BT}$

For law $\mathcal{L}_{BT}$, the control state has three protected terms *LawHashOne*, *LawHashTwo*, *Manager*. They are used to record the law hashes of Financial Department, Transport and Storage Department, and the address of Manager computer, respectively.

Rule $\mathcal{R}_1$ is the implementation of *newAdopted* event. Based on the hash of the law governing the actor, initial budget of a team member is assigned correspondingly. Moreover, the

Preamble:
law $\mathcal{L}_{TS}$ refines $\mathcal{L}_g$.

$\mathcal{R}1$.
```
UPON adopted(cert[issuer, subject, att]) DO
        -- verifying certificate, skipped ---
        -- initialize number of teams the employee joins ---
        add(teamNum(0))
```

$\mathcal{R}2$.
```
UPON joinRequest(conHost,commLaw, shortName) DO
         -- retrieve number of teams this employee has joined --
        teamNum(n)
        IF (n == 0) DO
        {
           -- allow the employee to join new team --
           -- adjust team number value in CS --
           forward(joinCommunity(-,commLaw, shortName),conHost)
           inc(teamNum)
        }
        ELSE DO
           deliver("You have joined one team")
```

Figure 8.14: $\mathcal{L}_{TS}$ Transport and Storage Department Law

number of purchase order is zero. Finally, a obligation named *PONumAdjustment* is imposed to re-initialize the value of the purchase number every 24 hours.

Rule $\mathcal{R}_2$ describes the *obligationDue* event dealing with the re-assignment of number of purchase orders (POs). Rule $\mathcal{R}_3$ is for the sending of a PO. The controller retrieves the budget as well as the law hash from the control state. The controller first check if the budget is enough to pay for the PO value *amt*. It also check if the number of POs made is still in range based on law hash. If everything is good, then the controller forwards the PO and reports to the manager. Otherwise, the controller drops the PO and informs the employee.

8.6 Related Work

The leading specifications for modeling service choreography, e.g., WS-CDL, BPMN [39][21], cannot be used to model choreographies, thus failing to solve their objective. The main problem is that they models choreographies by focusing on procedural aspects, e.g. by specifying control and message flow of the interacting services. Moreover, a choreography description in those models is not enforceable. Since a single misbehavior of an individual service may destroy the collaboration in a choreography, enforcement is a must to ensure safe and correct coordination

Preamble:

law $\mathcal{L}_{BT}$
authority(CA, keyHash)

– law_hash_one is hash of Financial Department Law –
protected(LawHashOne(hash1))

– law_hash_two is hash of Transport and Storage Department Law –
protected(LawHashTwo(hash2))

– record the address of Manager –
protected(Manager(host(redwine.rutgers.edu), port(3355)))

$\mathcal{R}1$.
```
UPON newAdopted(lh, arg[name]) DO
        IF (lh!="") DO
        -- actor is an LGI agent --
        {
           LawHashOne(lh1)
           LawHashTwo(lhTwo)
           -- initial budget of employee in Financial Dept --
           IF (lh == lh1) DO add(budget(1000))
           -- initial budget of employee in Transport Storage Dept --
           IF (lh == lh2) DO add(budget(800))
        }
        ELSE DO
            -- actor is a normal actor, i.e., parameter lh is "null" --
           add(budget(500))

        -- record the hash of law governing the actor --
        add(actor_lhash(lh))

        -- initial num of POs is zero --
        add(POperDay(0))

        -- re-initialize number of POs after 24 hours --
        imposeObligation("PONumAdjustment",60x60x24)
```

$\mathcal{R}2$.
```
UPON obligationDue(obl) DO
        IF (obl == "PONumAdjustment") DO
        {
           -- re-initialize number of POs made every day --
           remove(POperDay)
           add(POperDay(0))

           -- impose obligation to re-initialize number of POs --
           imposeObligation("PONumAdjustment",60x60x24)

        }
```

Figure 8.15: Law $\mathcal{L}_{BT}$ Buying Team Law

```
R3. UPON sent(_,PO(description, amt), dest) DO
          budget(bgt)
          LawHashOne(lh1)
          LawHashTwo(lh2)
          Manager(mgrAddress)
          POperDay(numPO)
          actor_lhash(lh)
          -- check if number of POs is in range based on employee's
          department --
          IF ((lh==lh1) AND (amt <= bgt)) OR
           ((lh==lh2) AND (amt <= bgt) AND (numPO < 10)) OR
           ((lh=="NULL") AND (amt <= bgt) AND (numPO < 3)) DO
          {
              dec(budget(bgt),amt)
              -- increase number of PO made --
              inc(POperDay(numPO))
              forward(_, PO(decsription, amt), dest)
              forward(_, PO(decsription, amt), mgrAddress)
          }
          ELSE DO
             deliver("Cannot place PO")
```

Figure 8.16: Law $\mathcal{L}_{BT}$ Buying Team Law (con't)

between interacting services.

To overcome these limits, we propose that choreography models should focus on specifying the constraints required to make all services collaborate correctly, without stating how such a collaboration is concretely carried out. We believe that services should have the autonomy to operate as well as they can, but they must satisfy the choreography constraints through enforcement. We extended the standard LGI mechanism for modelling and enforcing choreography constraints.

Chapter 9

Evaluation

In this section, we will present a number of results we have obtained in evaluating LG-SOA. This section addresses two issues: (a) processing time of events and operations which are supported by LG-SOA; and (b) a comparison between overhead incurred by LG-SOA and that incurred centralized coordination mechanisms (CCM).

9.1 Processing time of events and operations

We have employed a client and a server which host one service. The client and the server are located on the same LAN because we do not focus on communication time at this moment. To measure the number of requests the server can process without producing error, we gradually increase the rate the client sends request to the server. The maximum number of requests this particular server can handle is 125 requests per second. In a similar manner, we use the same server to host 9 other services, one at a time, and record the maximum number of requests each of them can handle per second with causing error. Note that even though we use the same server to host a single service, the *maximum requests per second* are different because they are different services providing different functionalities. Those data are presented in figure 9.1.

Now we place a controller in front of the server. For each service, the client sends requests at the corresponding rate determined in figure 9.1. We do this because we want the server to operate as if there is no controller between it and the client. When this request reaches the controller, an arrived event is fired and the controller forwards the request to the service. The service processes the request, and returns a response to the controller. This will triggers a sent event at the controller, and then the controller delivers the response to the client. At the client side, we record the time the client sends the request and the time it receives the response. The measured time reflects a roundtrip communication time between the client and the service, plus

Server	Maximum requests per second
SP1	125
SP2	158
SP3	235
SP4	82
SP5	130
SP6	190
SP7	110
SP8	90
SP9	70
SP10	120

Figure 9.1: Request processing rate

the time to evaluate arrived and sent events.

The event evaluation time depends on the event itself. For example, if we have a simple event say just record the time the request arrives at controller, the evaluate time is very small around 0.05 ms because there is no need to go through the content of the request. However, when we have a complex event like the one for processing ”ticket reservation” in chapter 6, the controller will have to check the request content and act accordingly, the evaluation time can go up to 1.5 ms. Generally speaking, the evaluation depends on the number of operations as well as the type of operations conducted. In this section, we only present the processing time for new operations/events that are only supported in LG-SOA. Other operations/events which are supported by both LG-SOA and standard LGI are presented in [8]. Figure 9.2 shows times to process them.

The above measurement has been conducted with client, T_i, and SP_i located on the following machines:

Generally speaking, the performance of a server SP_i depends on that of its controller T_i. Note that T_i has to process two events $arrived$ and $sent$, while SP_i only processes one request. Therefore, to maximize the performance of both SP_i and T_i, they should be given the same amount workload. In other words, best performance can be achieved if the time for SP_i to process a request is equal to the time T_i processes two corresponding events arrived and sent

Operation/Event	Processing time (ns)
deliverResponse	150
forwardRequest	180
searchST	130
grant	600
revoke	500

Figure 9.2: New Operation/Event for Client-Server Interaction under LG-SOA

	Machine Name	CPU Type	CPU Speed	Memory	Operating System
Client	rames	SunUltra10	440Mhz	256M	Solaris SunOS5.8
Controller	redwine	Intel686PIII	550Mhz	256M	Linux 1.6
Server	mco	IntelP4Dual	3.2 GHz	1G	Linux 2.6.8.121

Figure 9.3: Experiment Setting for Client-Server Interaction

for the request. We have verify this observation by placing the client, T_i and SP_i on three more settings, where each setting consists of three different machines that belong to in the Instructional Lab (I-Lab) domain at Rutgers University. Note that the processing time are different for different settings because machine configurations (CPU type, speed, memory, operating system) are different.

For the orchestration scheme, we need to consider 2 types of interactions: the first is between client and orchestrator, and the second is between an orchestrator and an orchestrated server. The first type is essentially the one we just presented. The second type only involves operations and events which are supported in standard LGI, so will not discussed further. The reader is referred to [8] for a detailed performance report of standard LGI.

For the choreography scheme, we employ two small programs which serve as two software actors. Each actor adopts a controller forming an agent which then adopts another controller in order to join a choreography. As shown in figure 9.4, we need 6 machines for this setup: two machines for two software actors, and each actor adopts two controllers running on two different machines.

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

Figure 9.4: Experiment Setting for Interaction under Choreography scheme

We have these two actors send messages to each other in a large number of times (10000). We then record the processing time of new operations/events that involve in this process in each time, and take the average. The data are presented in figure 9.5 below.

Operation/Event	Processing time (ns)
joinCommunity	550
newSendLG	150
event newAdopted	850
event joinRequest	650
event sendRequest	800
event newArrived	700

Figure 9.5: New Operation/Event for Choreography under LG-SOA

In Figure 9.5, for the case of an event the processing time is measured when the event is **empty**, i.e., there is no operation specified in the event. If there are operations defined in an event, then the total processing time is the sum of those of all operations plus that of the (empty) event itself.

We also do experiments on three more settings, each of which consists of 6 machines belonging to the Instructional Lab (I-Lab) domain at Rutgers University, but we do not present

the data here. The processing times are different for different settings because machine configurations (CPU type, speed, memory, operating system) are different. We do experiment on various settings to show that processing times for events and operations are negligible when comparing them with communication time.

9.2 Relative Overhead Under Various Conditions

We employ a model of overhead presented in [8] to show that overhead incurred by LGI is comparable to the overhead incurred by centralized coordination mechanisms (CCM) which use a conventional reference-monitor (like in Tivoli). To get a rough approximation for the behavior of the overhead of LG-SOA, comparing it to the overhead under CCM, we will use typical values for the quantities involved in them, ignoring many of the factors which may effect the overhead.

Typical communication times: These times depend on many factors, including the length of message, the communication protocol being used, the hosts involved, and the distance between the communicating parties. In [8]: the TCP/IP communication time within a LAN is measured as 5,000μs, the TCP/IP communication time within a WAN is 100,000μs, and the event evaluation time approximately is 50μs. Using a similar method we have the HTTP communication time within a LAN is 50,000μs and the HTTP communication time across WAN is 100,000μs.

9.2.1 Client-server interaction

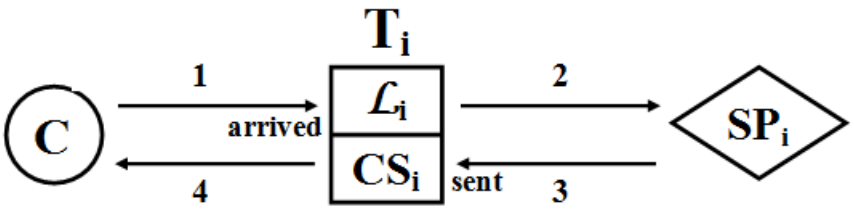

Figure 9.6: Client-Server Interaction under LG-SOA

For the client-server interaction shown in Figure 9.6, the time details for client C to send request to SP_i and receive a response are presented in figure 9.7

(Eq.1) Total Time = 4 * $T_{com} + 2 * T_{eval}$

Step	Communication time	Evaluation time
1	$T_{1,com}$	$T_{1,eval}$
2	$T_{2,com}$	
3	$T_{3,com}$	$T_{3,eval}$
4	$T_{4,com}$	

Figure 9.7: Time Details for Client-Server Interaction under LG-SOA

Where $T_{n,com}$ is the communication time in step n, $T_{n,eval}$ is the event evaluation time in step n; T_{com} is the communication time and T_{eval} is the event evaluation time in general.

The client-server interaction under centralized coordination (CCM) is shown in figure 9.8.

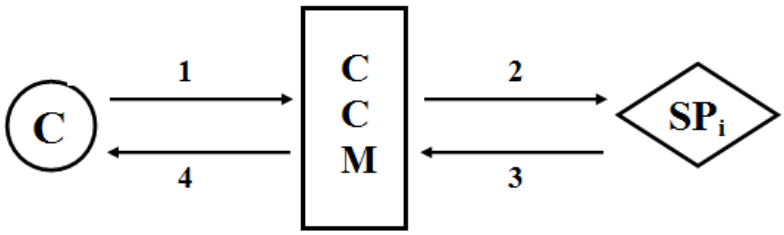

Figure 9.8: Client-Server Interaction under CCM

And, its time details are presented in figure 9.9.

Step	Communication time	Evaluation time
1	$T_{1,com}$	$T_{1,eval}$
2	$T_{2,com}$	
3	$T_{3,com}$	$T_{3,eval}$
4	$T_{4,com}$	

Figure 9.9: Time Details for Client-Server Interaction under CCM

(Eq.2) Total time = 4 * $T_{com} + 2 * T_{eval}$

Discussion:

- If SP_i is mediated by a controller in its own LAN, LG-SOA is better than CCM because:

(Eq.1) becomes (Eq.1') Total time = 2 * $T_{LAN} + 2 * T_{WAN} + 2 * T_{eval}$

From (Eq.1') and (Eq.2), we can see that client-server interaction under LG-SOA is much more efficient than under CCM because T_{WAN} is about 20 times of T_{LAN}.

- If client, controller, and server communicate with each other only via LAN (or WAN), LG-SOA and CCM have the same overhead.

9.2.2 Orchestration

Figure 9.10 shows interaction in orchestration, and time details are shown in figure 9.11

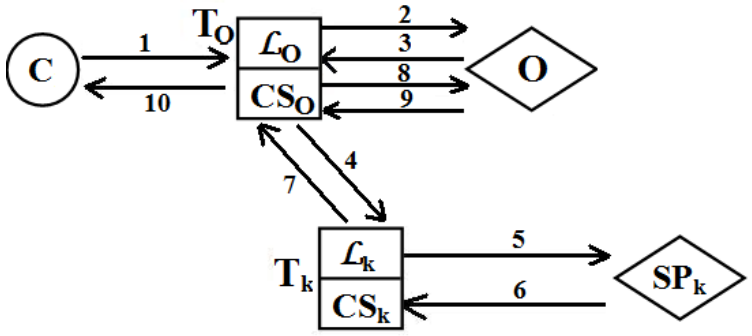

Figure 9.10: Orchestration under LG-SOA

Step	Communication time	Evaluation time
1	$T_{1,com}$	$T_{1,eval}$
2	$T_{2,com}$	
3	$T_{3,com}$	$T_{3,eval}$
4	$T_{4,com}$	$T_{4,eval}$
5	$T_{5,com}$	
6	$T_{6,com}$	$T_{6,eval}$
7	$T_{7,com}$	$T_{7,eval}$
8	$T_{8,com}$	
9	$T_{9,com}$	$T_{9,eval}$
10	$T_{10,com}$	

Figure 9.11: Time Details for Orchestration under LG-SOA

(Eq.3) Total Time = 10 * $T_{com} + 6 * T_{eval}$

Under CCM, orchestration interaction is shown in figure 9.12 and time details are presented in figure 9.13

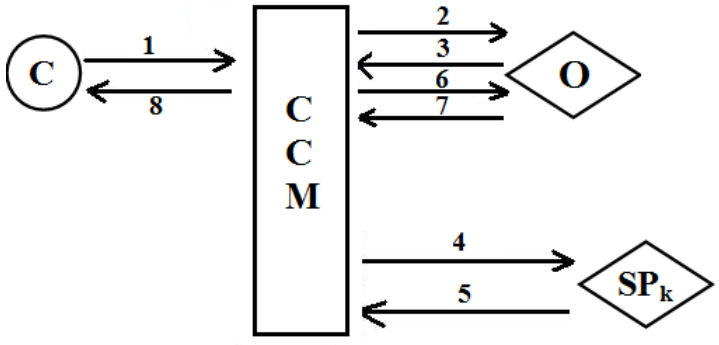

Figure 9.12: Orchestration under CCM

Step	Communication time	Evaluation time
1	$T_{1,com}$	$T_{1,eval}$
2	$T_{2,com}$	
3	$T_{3,com}$	$T_{3,eval}$
4	$T_{4,com}$	
5	$T_{5,com}$	$T_{5,eval}$
6	$T_{6,com}$	
7	$T_{7,com}$	$T_{7,eval}$
8	$T_{8,com}$	

Figure 9.13: Time Details for Orchestration under CCM

(Eq.4) Total Time = 8 * $T_{com} + 4 * T_{eval}$

Discussion:

- If O and SP_k are mediated by controllers in their own LANs and the two controllers T_o and T_k communicate via WAN, LG-SOA is better than CCM because

(Eq.3) becomes (Eq.3') Total time = 4 * $T_{WAN} + 6 * T_{LAN} + 6 * T_{eval}$

From (Eq.3') and (Eq.4), we can see that interaction under LG-SOA is more efficient than under CCM because T_{WAN} is about 20 times of T_{LAN}.

- If client, controllers, and servers communicate with each other only via LAN (or WAN), overhead incurred by CCM is less than that incurred by LG-SOA.

9.2.3 Choreography

Figure 9.14 shows an actor X sends a message to actor Y under LG-SOA, and its time details are shown in figure 9.15

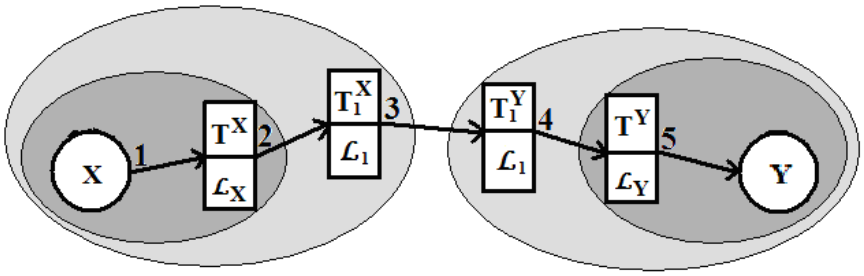

Figure 9.14: X sends a message to Y under LG-SOA

Step	Communication time	Evaluation time
1	$T_{1,com}$	$T_{1,eval}$
2	$T_{2,com}$	$T_{2,eval}$
3	$T_{3,com}$	$T_{3,eval}$
4	$T_{4,com}$	$T_{4,eval}$
5	$T_{5,com}$	

Figure 9.15: Time Details for Choreography under LG-SOA

(Eq.5) Total time = 5 * $T_{com} + 4 * T_{eval}$

Figure 9.16 shows interaction under CCM, and figure 9.17 shows time details accordingly.

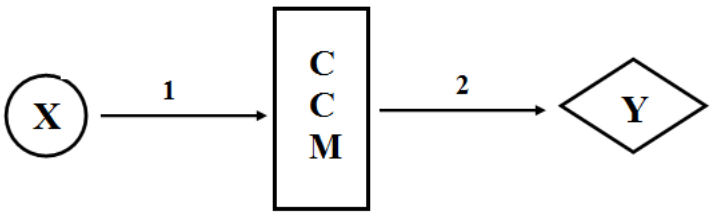

Figure 9.16: X sends a message to Y under CCM

(Eq.6) Total time = 2 * $T_{com} + T_{eval}$

Discussion:

- If X and Y are mediated by 2 controllers in their own LANs and the two controllers T_1^X and T_1^X communicate via WAN, LG-SOA is better than CCM because:

Step	Communication	Evaluation time
1	$T_{1,com}$	$T_{1,eval}$
2	$T_{2,com}$	

Figure 9.17: Time Details for Choreography under CCM

(Eq.5) becomes (Eq.5') Total time = $T_{WAN} + 4 * T_{LAN} + 4 * T_{eval}$

From (Eq.5') and (Eq.6), we can see that LG-SOA is more efficient than CCM.

- If X, Y, and their controllers communicate only via LAN (or WAN), overhead incurred under CCM is less than that incurred under LG-SOA.

Chapter 10

Conclusion

Under service oriented architecture (SOA), a software system consists of multiple heterogeneous servers, which may be distributed over the Internet, and may be managed under different administrative domains. SOA is an important realization of open systems in which software components may be added to or removed from the systems in an unpredictable manner. SOA has become hugely popular as the architecture of large and complex distributed systems, such as enterprise systems, grids, virtual enterprises, and supply chains. Unfortunately this architecture, as it is currently defined and being used, suffers from serious problems. In particular, it is not possible to ensure that a given system satisfies a desired global property, or to establish any regularity over a system. It is also not possible to ensure that commitments made by a given server to its clients are honored. It does not provide any methodology to address the concern of clients and orchestrated servers when they indirectly interact via an orchestrator. And, it is not possible to ensure that coordination between disparate servers—called ”choreography” under SOA— is carried out correctly.

LG-SOA –an extension of LGI– is a mechanism that addresses all these problems in a scalable manner. The most notable characteristics of LG-SOA are: (a) equip system managers with the capability to impose global constraints over a system, (b) give service providers the freedom to specify their commitments with clients while still enforcing global constraints across all services in the system, (c) address concerns from clients and orchestrated server when they indirectly interact with each other under orchestration scheme, (d) design a choreography model that focus on specifying the constraints required to make all services collaborate correctly, without stating how such a collaboration is concretely carried out; this model allows a service to join any choreography with no requirement about the order in which policies are specified.

We also presented how LG-SOA can be applied to legacy systems. Case studies in the

context of enterprise systems demonstrate the flexibility and applicability of this mechanism. Experiments show the overhead introduced by LG-SOA is relatively small, especially in the context of geographically distributed systems like SOA-based systems. In conclusion, LGI-SOA is effective and versatile in making SOA-based systems more coherent and trustworthy.

References

[1] X. Ao and N. Minsky. Flexible regulation of distributed coalitions. In the 8th European Symposium on Research in Computer Security (ESORICS), October 2003.

[2] X. Ao, N. Minsky, and T. Nguyen. A hierarchical policy specification language, and enforcement mechanism, for governing digital enterprises. In IEEE 3rd International Workshop on Policies for Distributed Systems and Networks Proceedings, June 2002.

[3] F. Casati, E. Shan, U. Dayal, and M.-C. Shan. Business-oriented management of web services. In Communications of the ACM Proceedings, October 2003.

[4] J. Clausing. Trade commission says geocities violated privacy rules. New York Times, August 1998.

[5] EFF. Aol's massive data leak. http://w2.eff.org/Privacy/AOL/, August 2006.

[6] EFF. Eff complaint to ftc regarding aol's massive data leak. http://w2.eff.org/Privacy/AOL/, August 2006.

[7] N. Minsky and C. Serban. Law Governed Interaction (lgi): A distributed coordination and control mechanism. http://www.moses.rutgers.edu/, October 2005.

[8] N. Minsky. LGI Reference Manual. http://www.moses.rutgers.edu/documentation/manual.pdf, September 2005.

[9] T. Ryutov and C. Neuman. Representation and evaluation of security policies for distributed system services. In DARPA Information Survivability Conference and Exposition, April 2000.

[10] J. Vitek and C. D. Jensen. Secure Internet Programming: Security Issues for Mobile and Distributed Objects. Springer, 1999.

[11] D. Kuo, A. Fekete, P. Greenfield, S. Nepal, J. Zic, S. Parastatidis, and J. Webber. Expressing and reasoning about service contracts in service-oriented computing. In IEEE International Conference on Web Services, September 2006.

[12] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In IEEE International Conference on Web Services, July 2004.

[13] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In IEEE International Conference on Web Services, September 2006.

[14] W. T. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang. Developing and Assuring trustworthy Web services. In International Symposium on Autonomous Decentralized Systems, April 2005.

[15] A. Betin-Can, T. Bultan. Verifiable Web Services with Hierarchical Interfaces. In IEEE International Conference on Web Services, July 2005.

[16] M.P. Papazoglou and D. Georgakopoulos. Serive-Oriented Computing. Communcications of the ACM, 46(10):2565, 2003.

[17] N. Minsky. Regularity-Based Trust in Cyberspace. International Conference on Trust Management. LCNS, May 2003

[18] T. Lam, N. Minsky. Enforcement of Server Commitments and System Global Constraints in SOA-based Systems. IEEE Asia-Pacific Services Computing Conference, December 2009.

[19] S. Godic, T. Moses. OASIS extensible access control markup language (XACML) version 2. Technical Report, OASIS, March 2005

[20] R. J. Hayton, J.M.Bacon, K. Moody. Access Control in an open distributed environment. IEEE Symposium on Security and Privacy 1998

[21] C. Ribeiro, P. Ferreira. A policy-oriented language for expressing security specifications. International Journal of Network Security, November 2007

[22] J.J.Jen, H.Chang, J.Y.Chung. A policy framework for web service based business activity management (BAM). Journal of Information Systems and e-Business Management, April 2004.

[23] N. Damianou, N. Dulay, E. Lupu, M. Sloman. The Ponder policy specification language. Policy Workshop, Bristol UK, January 2001.
[24] http://www.ebay.com
[25] http://www.napster.com/
[26] S. Marti, H, Garcia-Molina. Limited reputation sharing in P2P systems. 5th ACM conference on Electronic Commerce. New York, USA, May 2004.
[27] S. Kamvar, M. Schlosser, H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. The 12th International WWW Conference. May 2003.
[28] Xiong L, Liu L. PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities. IEEE Trans Knowl Data Eng 16(7):843857. 2004.
[29] B. Yu, M. Singh, K. Sycara. Developing trust in large-scale peer-to-peer systems. IEEE First Symposium on Multi-Agent Security and Survivability, pages 110, August 2004.
[30] Y. Liu, A. H. Ngu, and L. Zeng. Qos computation and policing in dynamic WS selection. In WWW, pages 66-73, 2004.
[31] Z. Xu, P. Martin, W. Powley, and F. Zulkernine. Reputation-enhanced qos-based web services discovery. In IEEE Internation Conference on Web Services 2007.
[32] Vu L-H, Hauswirth M, Aberer K. Qos-based service selection and ranking with trust and reputation management. Proceedings of 13th International Conference on Cooperative Information Systems (CoopIS 2005), October 31-November 4, 2005
[33] Griffiths N. Enhancing peer-to-peer collaboration using trust. Expert Syst Appl 31(4):849858. 2006
[34] J. Lyle. Trusted remote verification of web services. In TRUST, April 2009.
[35] H. Rajan and M. Hosamani. Tisa: Towards trustworthy services in a service-oriented architecture. In Services Computing, IEEE Transactions on Services Computing Proceedings, December 2008.
[36] V. Agarwal, P Jalote. Enabling End-to-End Support for Non-Functional Properties in Web Services. IEEE International Conference on Service Oriented Computing and Applications. 2009.
[37] C. Wang, S. Chen, and J. Zic. A Contract-based Accountability Service Model. 2009 IEEE International Conference on Web Services, Los Angeles, 2009.
[38] Qianxiang Wang, Jin Shao, Fang Deng, Yonggang Liu, Min Li, Jun Han, and Hong Mei. An Online Monitoring Approach for Web Service Requirements. IEEE Transactions on Services Computing. October 2009
[39] C. Schneider, F. Stumpf, C. Eckert. Enhancing Control of Service Compositions in Service-Oriented Architectures. IEEE International Conference on Availability, Reliability and Security, 2009.
[40] D. S. Rosenblum. A Practical Approach to Programming with Assertions. IEEE Transactions on Software Engineering, January 1995.
[41] T. Lam, N. Minsky. A Collaborative Framework for Enforcing Server Commitments, and for Regulating Server Interactive Behavior in SOA-based Systems. IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, November 2009.
[42] T. Murata and N. Minsky. Regulating Work in Digital Enterprises: a Flexible managerial Framework Cooperative Information Systems (CoopIS) Conference, Irvine California, October 2002.
[43] N.R. Jennings, P. Faratin, M.J. Johnson, P. O'Brien, and M.E. Wiegand. Using intelligent agents to manage business processes. First International Conference on The Practical Application of Intelligent Agents and Multiagent Agent Technology (PAAM96), pages 345-360, 1996.
[44] R. Medina-Mora, T.Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. ACM Conference on Computer Supported Cooperative Work, pages 281-288, November 1992.
[45] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language, Version 1.0. W3C Working Draft, December 2004.
[46] Z. He, T. Phan, and T. Nguyen. Enforcing enterprise-wide policies over standard client-server interactions. Symposium on Reliable Distributed Systems (SRDS), 2005.
[47] L. Gong and X. Qian. Computational issues in secure interoperation. IEEE Transctions on Software Engineering, pages 43-52, January 1996.
[48] C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. The 5th European Symposium on Research in Computer Security, pages 51-66, September 1998.

[49] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. IEEE Symposium on Security and Privacy, pages 66-80, May 2002.

[50] P. Bonatti, S. D. Vimercati, and P. Samarati. A modular approach to composing access control policies. The 7th ACM conference on Computer and communications security, pages 164-173, 2000.

[51] D. Wijesekera and S. Jajodia. Policy algebras for access control: the propositional case. The 8th ACM conference on Computer and communications security, pages 38-47, 2001.

[52] R. Kramer. iContract - The Java Design by Contract Tool. Technology of Object-Oriented Languages. August, 1998.

[53] A. Lazovik, M. Aiello, and M. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. The 2nd International Conference on Service Oriented Computing. ACM Press, New York, NY, USA, pages 94-104, 2004.

[54] E. M. Maximilien, M. P. Singh. Multiagent System for Dynamic Web Services Selection. International Conference on Autonomous Agents and Multi-Agent Systems, 2005.

[55] I. Jureta, S. Faulkner, Y. Achbany, M. Saerens. Dynamic Web Service Composition within a Service-Oriented Architecture. IEEE International Conference on Web Services, July 2007.

[56] M. Tanaka, T. Ishida, Y. Murakami, S. Morimoto. Service Supervision: Coordinating Web Services in Open Environment. IEEE International Conference on Web Services, July 2009.

[57] J. Parejo, P. Fernandez, A. Ruiz-Cortes, J.M. Garcia SLAWS: Towards a conceptual architecture for SLA enforcement. IEEE Congress on Services, Hawaii, 2008.

[58] M. Menzel, I. Homas, C. Meinel. Security Requirements Specification in Service-Oriented Business Process Management. The International Conference on Availability, Reliability and Security, 2009.

[59] M. Thomson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate-based access control for widely distrbuted resources. The 8th USENIX Security Symposium, August 1999.

[60] G. Karjoth. The authorization service of tivoli policy director. The 17th Annual Computer Security Applications Conference (ACSAC 2001), December 2001.

[61] H. Ludwig, A. Dan, and R. Kearney. Crona: An Architecture and Library for Creation and Monitoring of WS-agreements. International Conference on Service Oriented Computing. ACM Press, New York, NY, USA, pages 6574, 2004.

[62] S. Ponnekanti, and A. Fox. Interoperability among independently evolving web services. The 5th ACM/IFIP/USENIX international conference on Middleware. Springer-Verlag New York, Inc., New York, NY, USA, 331351, 2004.

[63] M. Rouached, O. Perrin, and C. Godart. Towards formal verification of web service composition. International Conference on Business Process Management. LNCS, vol. 4102. Springer, pages 257273. 2006.