A Collaborative Framework for Enforcing Server Commitments, and for Regulating Server Interactive Behavior in SOA-based Systems

Tin Lam, Naftaly Minsky
Department of Computer Science - Rutgers University
110 Frelinghuysen Road, Piscataway, NJ 08854
Email: {tinlt, minsky}@cs.rutgers.edu

***Abstract*—We are presenting a collaborative framework to enforce server commitments, and to regulate server interactive behavior in SOA systems. Trusted components collaborate to establish a network of enforcement components, one per server, to maintain proofs of what servers have committed with their clients and peer servers. The trusted components and the enforcement components also collaborate to make servers accountable for their commitments, and to enforce policies on the interactions between the servers and their clients/peer servers. In our framework, such collaborative efforts are explicitly stated in the form of policies to ease the management of collaborative processes. Our implementation is based on the LGI mechanism, complies with SOA standards, and does not require any change of the conventional SOAP-based protocol, the UDDI server, or the programs employed by servers and their clients.**

I. Introduction

There are several well-known incidents in which servers (service providers) have violated their privacy commitments with clients (service consumers). In August 1998, the website Geocities released its users' personal data to advertisers [1]. Another example, in August 2006, the search queries of around 650,000 users were disclosed [2] with the intention of being used by third-party researchers [3]. There are possibly many more violations that are still not known by the public. However, besides commitments announced *before* the occurrence of the client-server interaction such as privacy assurance or SLA, servers may make additional commitments with clients *during* their interactions. It is hard or even impossible for clients to determine whether such commitments would be satisfied. Therefore, clients need verifiable confidence that servers will abide by their commitments.

Moreover, the misbehavior or poor performance of a server may create worse effects to the whole system, compared to that of a typical client. Therefore, to protect the integrity of the whole system, it may be necessary to impose stricter constraints on servers. By continuously monitoring/auditing the interactive behavior of servers when they interacts with their clients and peer servers, trusted components, e.g. the programs employed by system administrators or accountants, can detect if servers violate system constraints, and react promptly to remedy the situation. Such remedies may be temporarily stopping poor-performance servers, or revoking server privilege of bad servers, to name a few.

To ensure that servers conform to policies, policies must be enforced *out of the control of servers*. There are a number of research projects that attempt to enforce policies on interactions between components in a system, e.g. [5], [4]. Unfortunately, they require the modification of existing applications. Since even a simple modification may result in many inter-related changes to a variety of software artifacts, people are not willing to adopt these approaches. Moreover, SOA systems may be large and heterogeneous, and servers may be dispersed throughout a distributed environment. Therefore, in SOA systems, system authorities need a systematic vehicle to *remotely* manage enforcement components and policies without requiring any change of existing applications. Finally, the fact that enforcement components are maintained and operate independently out of the control of the server helps convince clients of their neutrality and reliability.

We are presenting a collaborative framework to enforce server commitments, and to regulate server interactive behavior in SOA systems. Trusted components collaborate to establish a network of enforcement components, one per server, to maintain proofs of what servers have committed with their clients and peer servers. The trusted components also collaborate with the enforcement components to make servers accountable for their commitments, and to enforce policies on the interactions between the servers and their clients/peer servers. In our framework, such collaborative efforts are *explicitly stated* in the form of policies to ease the management of collaborative processes. Our technique employs Law Governed Interaction (LGI)-based middleware, and does not require any change of the conventional SOAP-based protocol, the UDDI server, or the programs employed by servers and their clients.

The rest of this paper is organized as follows: In section (II), we introduce an example of policies we seek to support. In section (III), we provide an overview of the LGI mechanism for SOA systems. We describe the implementation of our framework in section (IV), and demonstrate the use of our framework in a case study in section (V). A discussion about the implementation of our framework is presented in section (VI). We discuss related works in section (VII), and conclude in section (VIII).

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

II. Motivating example

Assume in an SOA system, besides servers and clients, there is a system administrator SA, an accountant $Acct$, a certification authority CA, and a UDDI server. All of them (except servers and clients) are considered as trusted components in the system.

A. Grant/Revoke server privilege

We consider providing services as server privilege. Suppose there is a policy concerning server privilege as follows.

> A1. A server must present to SA a certificate signed by CA to authenticate that it (1) has a particular name $SPname$, (2) provides services at the endpoint EP, and (3) operates under a policy P. After successful authentication, SA will grant the privilege of providing services to the server. How CA issues such a certificate to the server is beyond the scope of this paper.
>
> A2. If SA revokes the server privilege of a server, the server must stop providing services to clients in the system.

B. Retain server privilege

There are certain requirements a server must satisfy to retain the server privilege. Below is an example of such requirements:

> B1. A server must report its average response time (ART) to $Acct$ every 60 minutes.
>
> B2. Each server must provide audit information about interactions with its clients if it receives a command $initAudit$ from $Acct$. The audit information includes the name of the service to be invoked, the time the request is received, and the time the corresponding result is returned. The audit process is stopped when the server receives a $endAudit$ command issued by $Acct$.
>
> B3. A server is allowed to invoke services provided by other servers up to h times within an hour.

The requirement (B3) above prevents a server from abusing services. To enforce this requirement, for each server, the system must maintain a counter which saves the total number of services invoked by the server across the entire system.

C. Server policy

A server's policy 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. The interaction between a server and a client may involve a sequence of messages exchanged in a predefined order, called conversation [6]. 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. Given a client's request, a server can determine the client's sensitive data which may include (1) the 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.

The second type of server commitments is made by the server *during* a conversation, and the server generally promises that it will satisfy those commitments. Since the terms exchanged between the server and the client are *unknown before* their conversation starts, there is a need to record important notes of the conversation. If a dispute occurs, the conversation records serve as proofs of what the server promised, and will make the server responsible for their words.

Suppose there is a "travel agent" server which supports a "ticket selling" conversation. Below is an example of a server policy consisting of both requirements and commitments.

> C1. Client Identification: A client has to submit its certificate signed by CA to identify itself when it reserves or buys a ticket from the server.
>
> C2. Commitment before conversation starts: the server will not release the client's certificate and IP address of the client's application to other third parties.
>
> C3. Commitment during a conversation: during their conversation, a client may request to reserve the right to buy a ticket at a particular price p with a grace period g. If the server agrees, it is obliged to sell this ticket to the client for p, if the client pays for the ticket within period g. Moreover, the server commits itself to not sell the reserved ticket to anybody else before the end of the grace period.

Note that in the scenario above, the server needs to *link messages belonging to the same conversation* in order to determine who have reserved/bought a particular ticket.

D. Discussion

The requirements in (A) and (B) affect all servers in the system, while (C) is only applicable to one server. Therefore, (A) and (B) can be considered as system constraints which will be expressed in a global policy; and (C), expressed in a server policy, are specific constraints applied to one particular server.

Servers are not trusted in the sense that they do not *always* conform to policies. For example, a server may not voluntarily give up the server privilege as commanded by SA, or a server may report wrong audit information to $Acct$. Such examples show that policies must be enforced *out of the control* of the server.

To enforce (A1) and (A2) in our framework, SA collaborates with the UDDI server to assign an enforcement component for a server. The server has to interact with other components in the system through its enforcement component. In other words, getting the server privilege means the server's interactive behavior is under supervision of its associated enforcement component. SA revokes the server privilege by collaborating with the UDDI server to revoke the enforcement

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

component of the server. This collaboration is described in section (IV.B).

To enforce (B1),(B2), and (C3) the enforcement component collaborates with *Acct* by reporting truthful data about interactions between the server and its clients. Note that the enforcement component can enforce (C1) and (C2) on its own. How the enforcement component intercept the client-server interaction is described in section (IV.C).

To enforce (B3), two enforcement components of the two involved servers need to collaborate. This collaboration is explained in more details in section (IV.D).

III. An Overview of LGI mechanism for SOA

LGI is a coordination and control mechanism for heterogeneous distributed systems. The reader is referred to [14] for a detailed description of LGI. We have modified the standard LGI mechanism to make it work for SOA systems. Specifically, each server *ser* is associated with an enforcement component, called *controller* $T_{\mathcal{L}}$, which is trusted to enforce a policy $\mathcal{L}$ on the interaction between *ser* and its clients. In our framework, a policy is expressed as an LGI law maintained by a law server. From now on, we use the terminologies "law" and "law server" instead of "policy" and "policy server".

The purpose of $\mathcal{L}$ is to decide what should be done in response to the occurrence of certain regulated events, such as the receipt or the sending of a message at $T_{\mathcal{L}}$. This mandated response, called the ruling of $\mathcal{L}$, is a function $\mathcal{L}(E, CS)$, where CS is the control state of $T_{\mathcal{L}}$ at the time of the occurrence of the event E. Such a ruling is a sequence of zero or more *control operations*, which can cause such things as forwarding of messages, and updating the control state CS of $T_{\mathcal{L}}$.

Under LGI mechanism for SOA, a controller has a name "server@machine-name". For example, a controller started on the machine "cornpops.rutgers.edu" is identified as "server@cornpops.rutgers.edu". Note that we assume there is only one controller running on a machine.

An LGI law consists of a sequence of events. Each event is expressed as a *rule* of the form $UPON(event)$ $IF(condition)$ $DO(action)$ where $(condition)$ is a general expression defined over the event and the control state CS of $T_{\mathcal{L}}$, and $(action)$ is one or more operations mandated by the law. We introduce here briefly the definition of four regulated events which will be demonstrated in our case study in section (V).

---

adopted(): the first event in the life of a controller.
sent(m): $T_{\mathcal{L}}$ receives a response m sent by the server *ser*.
arrived(m): a request m of a client arrives at $T_{\mathcal{L}}$.
arrived(src, m, dest): message m is forwarded from controller "src" to "dest", where "src" and "dest" are LGI names of the two controllers.
obligationDue(obl): an obligation named *obl* is fired. Typically, obligations are set to occur sometime in the future to ensure that some actions are carried out in a timely manner.

The body of a rule may contain two distinguished types of operations. The first type has the form $t@CS$, where t is any Prolog term. It checks to see if the term t exists in the control-state of $T_{\mathcal{L}}$. The second one is a primitive-operation. Commonly used primitive operations include:

---

add(s(v)): $T_{\mathcal{L}}$ adds a term s with value v to its cs.
remove(s): $T_{\mathcal{L}}$ removes a term s from its cs.
replace(T1, T2): $T_{\mathcal{L}}$ replaces $T1$ in its cs by $T2$.
release(m, host, port): $T_{\mathcal{L}}$ sends a message m to an application which is waiting for messages at $(host, port)$.
forwardRequest(m): $T_{\mathcal{L}}$ forwards message m to *ser*.
forwardResponse(m): $T_{\mathcal{L}}$ forwards message m to a client.
forwardNext(m, Tn) $T_{\mathcal{L}}$ forwards its server's request to Tn, the controller of another server. This operation will cause the firing of event **arrived(src, m, dest)** at controller Tn.
grant(SPName,EndPoint,lawURL): $T_{\mathcal{L}}$ must be ready to enforce law of a server whose name is "*SPName*", endpoint is "*EndPoint*", and law text is available at "*lawURL*". We will use this operation to assign a controller to a server. This is a controller management interface. More details about controller management interfaces are presented in section (IV.A).

An http server, called law server, is responsible for maintaining all law texts, and making them available online. A complete list of LGI regulated events and primitive operations is available in [10].

Law Hierarchy

Consider a hierarchy, or tree, of laws $H(\mathcal{L}_0)$, rooted at a given law $\mathcal{L}_0$. Each law in the hierarchy can define the manner in which it can be refined. Specifically, the presence of a $delegate(g)$ statement in a law invites a *law refinement* to propose operations to be added to the ruling being computed. Consider a law $\mathcal{L}$ with the following rule r:

UPON event DO ..., delegate(g), ...

If a law $\mathcal{L}'$ is *directly subordinate* to $\mathcal{L}$, then every evaluation of a ruling of $\mathcal{L}'$ will start with the rules in $\mathcal{L}$. If this evaluation gets to the $delegate(g)$ statement of the rule r above, then goal g is submitted to the law refinement $\overline{\mathcal{L}'}$ for evaluation. This evaluation will produce a set of operations - we call this set the *ruling proposal* of $\overline{\mathcal{L}'}$ for goal g.

The structure of law refinement: A law refinement component $\overline{\mathcal{L}'}$ of a law $\mathcal{L}'$ from $\mathcal{L}$ looks pretty much like the root-law $\mathcal{L}_0$, with two distinctions. First, the top clause in the component is *law L' refines L*. Second, the names of the rules in $\overline{\mathcal{L}'}$ need to match the goals delegated to it by $\mathcal{L}$.

Rewriting a ruling proposal of a law refinement: $\mathcal{L}$ can regulate the effect of a law refinement on the eventual ruling of the law $\mathcal{L}$ by rewriting the ruling proposal. Specifically, the $rewrite$ rules in $\mathcal{L}$ determines what is to be done with each operation proposed by a law refinement: whether it should be blocked, included in the ruling, or replaced by some list of operations.

Finally, a statement $protected$ can protect a term in the control state from modification by law refinements. For example, if the statement *protected(ID( ), nick( ))* appears in the "preamble" of $\mathcal{L}$, then no refinement of $\mathcal{L}$ can propose an

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

operation that modifies terms of the form (ID, nick). Strictly speaking, such protection can be carried out via $rewrite$ rules, but the "protected" statements are more convenient for this purpose.

On the effects of cascading delegation: To this point, our discussion has focused on the interaction between a law and its *immediate* law refinement. Consider now a chain of law refinements, where $\overline{\mathcal{L}_1}$ refines a root-law $\mathcal{L}_0$ to form $\mathcal{L}_1$, $\overline{\mathcal{L}_2}$ refines $\mathcal{L}_1$ to form $\mathcal{L}_2$, and so on. The invocation of a $delegate$ clause in $\mathcal{L}_0$ can lead to the invocation of a $delegate$ clause in $\overline{\mathcal{L}_1}$, which in turn can lead to the invocation of a $delegate$ clause in $\overline{\mathcal{L}_2}$, and so on. Suppose this process eventually stops in $\overline{\mathcal{L}_m}$, where a $delegate$ is not invoked as part of the ruling. Then, $\overline{\mathcal{L}_m}$ will eventually return a ruling proposal to $\mathcal{L}_{m-1}$, which will be subjected to $rewrite$ rules in $\mathcal{L}_{m-1}$. Eventually, $\mathcal{L}_{m-1}$ will return a ruling proposal to $\overline{\mathcal{L}_{m-2}}$, which will be subjected to $rewrite$ rules in $\mathcal{L}_{m-2}$. This process repeats until $\overline{\mathcal{L}_1}$ returns a ruling proposal to $\mathcal{L}_0$, where it will stop.

Complete specification of LGI law hierarchy and its demonstration can be found in [8], [9].

IV. Framework Overview

Figure (1) gives an overview of our framework. This figure consists of one server, one UDDI server, CA, SA, $Acct$, and one law server. However, our framework is applicable to any system which has multiple servers. We use the term SP_i to refer to a server in the system, $\mathcal{L}_i$ is the law of the server, and T_i is the controller of the server.

We use a two-level LGI law hierarchy to organize laws in our framework. The global policy and the server policy discussed in section (II.D) are expressed in $\mathcal{L}_g$ and $\mathcal{L}_i$, respectively. $\mathcal{L}_g$ is on the top level, and $\mathcal{L}_i$ is on the second level. $\mathcal{L}_i$ is formed by refining $\mathcal{L}_g$ with $\overline{\mathcal{L}_i}$. The interaction between a client, T_i, and SP_i is regulated according to $\mathcal{L}_i$.

A. 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 $\mathcal{L}_{gen}$ is beyond the scope of this paper. We, however, present an example of the generic law in section (V) 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.

B. Grant/Revoke Server Privilege

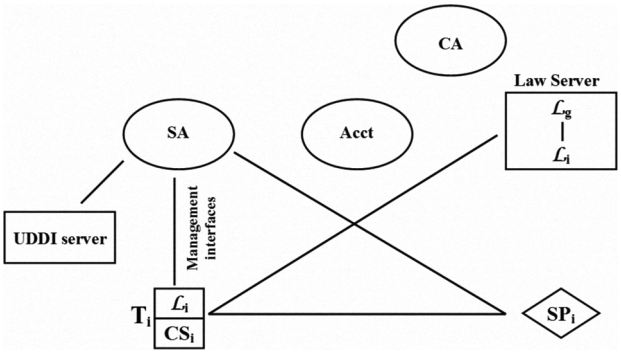

Fig. 1. Framework Overview

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 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.

SA collaborates with the UDDI server to assign a controller for SP_i. This collaboration is described in a law called $\mathcal{L}_{sa}$. Specifically, SA is a controller operating under law $\mathcal{L}_{sa}$ (for "system administration" law). That law instructs how to process a "ServerPrivilege" request from SP_i, as described above.

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).

In our framework, 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

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

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 expect that our approach is easily integrated into any existing SOA system.

Finally, $\mathcal{L}_g$ defines who can use the "revoke" management interface. This ensures the "revoke" command is applied to all servers under the same condition. Laws $\mathcal{L}_{gen}$, $\mathcal{L}_{sa}$, $\mathcal{L}_g$, and $\mathcal{L}_i$ are shown in section (V).

C. T_i handles a client's request

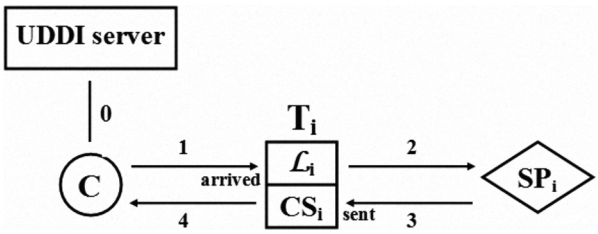

Fig. 2. T_i handles a client's request

The interaction between a client c, T_i, and SP_i is presented in figure 2. 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 framework 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.

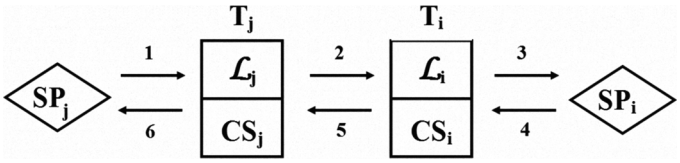

Fig. 3. Dual mediation

D. Dual mediation

Figure 3 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, our framework makes use of 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. The requirement (B.3) described in section (II.B) is implemented by dual-mediation, as shown above. The implementation of (B.3) is shown in section (V.C).

V. Case study

In SOA systems, three main components (UDDI server, client, server) communicate with each other by exchanging SOAP messages. So, we also have SA and $Acct$ communicate with controllers by SOAP messages.

There is an *implicit* constraint in $\mathcal{L}_g$. That is, for each command in the set $\{initAudit, endAudit, revoke\}$ 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.

While interacting with its clients, a server 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 $\overline{\mathcal{L}}$.

Law $\mathcal{L}_{gen}$ and $\mathcal{L}_{sa}$ shown below are just for demonstration purpose. They can be more complex in real scenarios.

A. Generic Law

Law $\mathcal{L}_{gen}$ is shown in figures 4. Its "preamble" specifies that $\mathcal{L}_{gen}$ accepts certificates from CA represented by keyHash.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

```
Preamble:
  law L_gen
  authority(CA, keyHash)

R1)
  UPON 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)
         }
```

Fig. 4. Law $\mathcal{L}_{gen}$ - Generic Law

$\mathcal{L}_{gen}$ 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 $lawURL$.

B. Law $\mathcal{L}_{sa}$

```
Preamble:
  law L_sa
  authority(CA, keyHash)

R1)
  UPON arrived([reqID, s, reqcnt]) DO
       cert = SearchST("Cert", reqcnt)
       IF(cert[issuer(CA),subject(SP),attr(EP,
       lawURL])DO
         IF (s=="ServerPrivilege")
         {
          T=getController()
          forwardNext([grant(SP,EP,lawURL)],T)
          updateUDDI(SP,T,lawURL)
         }
```

Fig. 5. Law $\mathcal{L}_{sa}$ - System Administration Law

Law $\mathcal{L}_{sa}$ is shown in figures 5. Its "preamble" specifies that $\mathcal{L}_{sa}$ accepts certificates from CA represented by keyHash. Recall that SA is a controller operating under law $\mathcal{L}_{sa}$. Rule $\mathcal{R}_1$ of $\mathcal{L}_{sa}$ describes how SA processes a "ServerPrivilege" SOAP message sent from a server. First, SA verifies the server's certificate. If the certificate is valid, it will choose a generic controller T. Then, it uses the operation "forwardNext" to send a "grant" command to T. This will cause the firing of event "arrived" at T, which is operating under $\mathcal{L}_{gen}$. T becomes T_i (as defined in the generic law $\mathcal{L}_{gen}$ in figure 4). Finally, SA publishes the server's description to the UDDI server using T's address as the endpoint. For the sake of simplicity, the code of the function "updateUDDI" is not shown here.

```
Preamble:
  law L_g
  authority(CA, keyHash)
  protected(Inv(req(_),ser(_),t(_)))
  protected(numOfInv(_))
  protected(totalPT(_))
  protected(Acct(host(_), port(_)))
  protected(InvPerH(_))


R1)
  UPON adopted() DO
       imposeObligation("ReportART", 60x60)
       imposeObligation("InvReset", 60x60)
       add(Acct(host(host), port(port)))
       add(InvPerHr(0))

R2)
  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
          // a "revoke" management interface
          IF (s =="revoke") DO grant(_,_,genlaw)
       }
       ELSE DO
       {
        cert = SearchST("Cert", reqcnt)
        IF (cert[issuer(CA),subject(Self)]) DO
        {
            // request from the server
           InvPerHr(k) @ CS
           IF (k < 1000) DO
           {
             inc(InvPerHr, 1)
             Tn= SearchTn(reqcnt)
             reqcnt = RemoveTn(reqcnt)
             forwardNext(reqcnt, Tn)
         }
          ELSE DO
            forwardResponse("Limit reached")
        }
       }
       ELSE DO // a normal client's request
       {
        add(Inv(req(reqID),act(s),T(curTime)))
        delegate(arrivedCont,[reqID,s,reqcnt])
       }

R3)
  UPON arrived(peer,[reqID,s,m,flag],Self) DO
       IF (flag=="next") DO
       {
         //m is a request from another server
        add(Inv(req(reqID), act(s), T(curTime)))
        delegate(arrivedCont, [reqID, s, m])
       }
       ELSE DO
       {
        //m is a response from another server
        //to answer a request of this server
         forwardResponse(m)
       }
```

Fig. 6. Law $\mathcal{L}_g$ - Global Law

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

```
R4)
 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)
R5)
 UPON obbilationDue(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)
      }
      IF (obl == "InvReset") DO
      {
         remove(InvPerHr)
         add(InvPerH, 0)
         imposeObligation("InvReset",60x60)
      }
```

Fig. 7. Law $\mathcal{L}_g$ - Global Law (continued)

C. Global Law $\mathcal{L}_g$

Law $\mathcal{L}_g$ is shown in figures 6 and 7. Its "preamble" specifies that $\mathcal{L}_g$ is a root law, and accepts certificates from CA represented by keyHash. The control state has five protected terms *'inv(req(reqID), act(s), T(t))'*, *'numOfInv(num)' 'totalPT(pt)'*, and *'Acct(host(h), port(p))'*, *'InvPerH(h)'*. 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, the (host/port) of $Acct$'s application, and the number of service invocations requested by SP_i, 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 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. Another obligation "InvReset" is also imposed to reset the number of service invocations of the server every hour.

Rule $\mathcal{R}_2$, the *first* $\mathcal{L}_g$'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. The calling of the primitive operation "grant(-,-,genlaw)" signifies that a controller revocation command is issued, and it is the time T_i is back as a generic controller and works under the generic law $\mathcal{L}_{gen}$.

Otherwise, if this is a request from its associated server SP_i for a service provided by SP_j, T_i checks whether SP_i has reached the invocation limit (1000 invocations) : if "yes", T_i returns a message "Limit reached" to SP_i; if "no", T_i extracts and removes the *"SOAP intermediary"* value (called **Tn**) kept in the SOAP request header, and forwards the modified SOAP request to **Tn** using the operation "*forwardNext*". Recall that **Tn** is the controller of SP_j. If this is the request from a client of SP_i, 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 $\mathcal{R}_3$, the *second* $\mathcal{L}_g$'s **arrived** event, is fired when a message m is forwarded from a peer controller (called "peer") to T_i (called Self). Based on the parameter *"flag"*, T_i checks if m is a *forwarded request* from T_j, i.e. SP_j wants to invoke a service of SP_i and T_j has used the operation "*forwardNext*" to forward the request to T_i. If this is the case, T_i will treat m as a request from a normal client. Otherwise, i.e. m is a response sent back from SP_j through T_j, T_i will forward m to SP_i.

Rule $\mathcal{R}_4$, $\mathcal{L}_g$'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 $\mathcal{R}_5$ is discharged when the obligation "ReportART" is due. This obligation requires T_i to report SP_i's ART to $Acct$ every 60 minutes (60 x 60 seconds). Another obligation "ReportART" is set to fire in the next 60 minutes. Similarly, the obligation "InvReset" is fired each hour to reset the number of service invocations of SP_i, and T_i re-imposes this obligation.

D. Server Law $\mathcal{L}$

A conversation between SP and its client involves three services: "*asks*", "*reserves*", and "*pays*". The client c invokes the "*asks*" service to query SP about available tickets (uniquely identified by *tkIDs*) and their corresponding prices. We view SP'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 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. T does **not** influence SP regarding how to deal with c's proposal. Therefore, if SP agrees, SP must be **accountable for its decision**. c uses service "*pays(tkID)*" to pay for its reserved ticket.

$\overline{\mathcal{L}}$ is shown in figure 8. $\overline{\mathcal{L}}$'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 saves a term

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

```
Preamble:
  law L refines L_g
  authority(CA, keyHash)

R1)
 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
          forwardResponse("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
         forwardResponse("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)
        }
       }
      }

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)))
       }
      }
      forwardResponse(respcnt)
```

Fig. 8. Refinement $\overline{\mathcal{L}}$

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 for available tickets, T simply forwards c's inquiry to SP. In both "reserves" and "pays" cases, T 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 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 returns to c the message "Somebody else has reserved this ticket" and drops c's request. This way, T *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 calls the function $searchST("ProposedPrice", reqcnt)$ to extract the value *pp* of the field *ProposedPrice* in c's SOAP request *reqcnt*. T 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 $\overline{\mathcal{L}}$), and forwards c's request to SP.

Consider the case when c invokes "*pays(tkID)*". If c's certificate is valid and its reservation is still effective, T will **sell the ticket to the client on SP's behalf**. Specifically, T 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 reports c's receipt to $Acct$. Finally, T forwards the request to SP. Note that T cannot control what SP would actually do with the forwarded request "*pays(tkID)*", but T 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 and $Acct$ will support c if any dispute arises.

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

Note that we also implements the privacy assurance in $\mathcal{R}1$ of $\overline{\mathcal{L}}$. The main idea is to generate a unique nickname for each client and keep a pair(client, nick) in CS. Each time T 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 replaces the client's certificate in the SOAP request by $nick$ and forwards the *modified* request to SP; otherwise, T 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. This way, even though SP does not receive the client's certificate, SP is still able to determine if the request is from a returned client based on the client's nickname. In $\mathcal{R}1$ of $\overline{\mathcal{L}}$, we uses the function "findNick" to deal with the searching/creating a nickname for a given client.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

VI. Discussion

A. Controller Service

To support large numbers of servers, which may operate under a variety of laws, there is a need to provide a large set of *generic controllers* that can be widely trusted to operate in compliance with any valid law loaded into them. Such a collection of controllers would serve the role of a trusted computing base (TCB). But unlike the traditional TCB, which is usually centralized, our collection of controller is designed to be decentralized, and is thus referred to as decentralized TCB, or DTCB. Such a DTCB needs to be provided by a reliable service, called a controller service (CoS), which creates, maintains, continuously tests, and certifies a distributed collection of controller.

Such a CoS can be operated by an enterprise for its own internal use; by a federation of enterprises for the use of its members; or by a regional authority, such as a municipality, for the use of servers in its range. The current implementation of LGI provides an experimental version of a CoS. And there is an ongoing research on techniques for protecting the controllers maintained by a CoS from various kinds of attacks; in particular, by intrusion detection, by making it harder to target specific servers, or specific law; and by other means.

It should be pointed out that if our framework is to be used by arbitrary servers all over the Internet, then the CoS needs to be managed by a reputable commercial company or governmental institution, whose business it is to provide its customers with trustworthy controllers. This organization must be willing to vouch for the trustworthiness of its controllers, and to assume liability for their failures. Such an Internet wide CoS is yet to be established.

B. Client Cooperation Prevents Servers From Cheating

In our current framework, for the sake of simplicity we assume only SA can publish a server description to the UDDI server. This way, SA can assure that only *registered servers are made known to clients through the UDDI server*. Since this assumption is described in laws, it it straightforward for system designers to write laws to allow other principals to access to the UDDI server. Such laws serve as access control policies to the UDDI servers.

It is possible that a client may know a server's physical endpoint, and sends a request directly to the server. In that case, since messages exchanged between the client and the server are not regulated by the controller, there is no guarantee that the server's law is enforced. Here, our framework requires the client's cooperation to make its interaction with the server secure. The client cooperates by using the server's description retrieved from the UDDI server. In our point of view, this is a reasonable requirement because generally clients would act to secure their benefits. If this requirement is satisfied, the client-server interactions are regulated by controllers, and servers are not able to cheat.

VII. Related works

Some approaches have recently been proposed to verify whether a service implementation conforms to its service-level agreement, as seen in the works of [12], [13]. These two projects share our goal to provide assurances to clients that servers in SOA systems honor their commitments. Both of these provide means for verifying that certain promises made by a server are actually satisfied. The verification is done by *inserting assertions into the code of the server program.* Although both projects employ a Trusted Platform Module architecture, the clients of a given server cannot be really sure that the correct verification has been done, because this verification is *very specific to the code of the server*, which is generally unknown to the clients. In our framework, server commitments are explicitly stated in laws, which can be examined by clients.

Other projects, e.g., [16][17][18], focus on verifying the service behavioral contracts which are defined by the visible interface of services. In [16], boolean formula associated with messages help verify if a given message exchange is legal. The works [17], [18] uses run-time monitoring to validate the functional and non-functional requirements of web services. However, in these works, non-functional requirements like (C2) and (C3) presented in section (II) are not addressed.

There are several research projects which address the testing phase of services, such as [19][20]. 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 they specified?". Note that passing the test phase does not mean a server would always operate as it committed.

The use of law hierarchy allows us to impose global constraints on the interactive behavior of servers. Global constraints ensures that certain system properties are met, and in our opinion, is very necessary for SOA systems which may consist of multiple heterogeneous servers, and may be managed under different administrative domains. However, imposing global constraints is not considered in the research projects mentioned above. Moreover, our framework provides a systematical vehicle to remotely manage the network of controllers according to explicitly stated laws. This is a major technical advantage when we need to deploy enforcement components for servers which may be distributed over a wide area of network.

This work is a significant extension of our previous one [11] in the following aspects. First, we introduce and implement the concept of using policies to specify how trusted components collaborate to establish a network of LGI controllers to enforce server commitments. Second, we show how LGI controllers collaborate with the trusted components mentioned above to regulate server interactive behavior which also includes server-server interaction, a common interaction in SOA systems

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347

which is not considered in [11]. In this interaction, the two controllers of the two involved servers dual-mediate messages exchanged between a server and its peer server. Finally, all collaborative actions are explicitly specified in LGI laws, which makes the management of collaboration easier.

VIII. Conclusion

We propose a framework in which trusted components collaborate to create a network of enforcement components, one per server, to maintain proofs of what servers have committed with their clients and peer servers. In turn, those enforcement components collaborate with the trusted components mentioned above to enforce server commitments, and to regulate the interactive behavior of servers. Server commitments and collaborative actions are explicitly stated as LGI laws, and are strictly enforced by LGI mechanism.

We demonstrate the feasibility of our framework through a case study. Our framework supports standards in SOA systems and does not require any change to existing applications in the system or to the SOAP-based protocol. We conclude that our framework is practical and easy to integrate into existing SOA systems.

References

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

[2] EFF. Aols massive data leak. http://w2.eff.org/Privacy/AOL/, August 2006.

[3] EFF. Eff complaint to ftc regarding aols massive data leak. http://w2.eff.org/Privacy/AOL/, August 2006.

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

[5] J. Vitek and C. D. Jensen. Secure Internet Programming: Security Issues for Mobile and Distributed Objects. Lecture Notes in Computer Science. Springer-Verlag, 1999.

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

[7] N. Minsky and V. Ungureanu. Law-governed interaction: A coordination and control mechanism for heterogeneous distributed systems. In ACM Transactions on Software Engineering and Methodology (TOSEM), July 2000.

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

[9] 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.

[10] N. Minsky. Law governed interaction (lgi): A distributed coordination and control mechanism. http://www.moses.rutgers.edu/, October 2005

[11] T. Lam and N. Minsky. Enforcement of Server Commitments and System Global Constraints in SOA-based Systems. In IEEE Asia-Pacifc Services Computing Conference (IEEE APSCC 2009), December 2009.

[12] J. Lyle. Trustable remote verification of web services. In 2nd International Conference on Trusted Computing, April 2009.

[13] H. Rajan and M. Hosamani. Tisa: Towards trustworthy services in a service-oriented architecture. In IEEE Transactions on Services Computing Proceedings, December 2008.

[14] N. Minsky. Law governed interaction (lgi): A distributed coordination and control mechanism. http://www.moses.rutgers.edu/, October 2005.

[15] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In IEEE Symposium on Security and Privacy, May 1997.

[16] 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.

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

[18] 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.

[19] 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.

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

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8347
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8347