A new authorization model and its mechanism using service paths in open distributed environments

Masakazu Soshi and Mamoru Maekawa
Graduate School of Information Systems, University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo, JAPAN 182
E-mail: {soshi,maekawa}@maekawa.is.uec.ac.jp

Abstract

In open distributed systems, multiple software agents (or objects) work in cooperation to achieve some goal. Therefore, we need an authorization model that can control security and trust relationships of agents. *Saga Security System* does offer such an auhorization model as well as a security mechanism, both of which are thoroughly discussed in this paper.

In the model, the unit of authorization is a *service context*, which represents a runtime state of an operation. Contrary to expectation, however, authorization to invoke a service context is not given to a principal as usual, but *to a service path*, which is defined as a sequence of service context invocations. This approach is one of the outstanding features of the model and provides flexible and uniform protection suitable for object-oriented systems and cooperative agent systems. Additionally, we present the security mechanism of Saga Security System. Security of a Saga Agent during its traversal is controlled by the Security Monitor integrated with the Agent.

Keywords

security architecture, authorization model, security mechanism, distributed system, mobile agents or objects.

1 INTRODUCTION

In recent years, inexpensive and high performance computers come into wide use, and more and more computers become interconnected through computer networks. One of the greatest advantages of such open environments is easy sharing of computer resources – but one catch. The ease of sharing in the environments naturally leads to the difficulty of ensuring security.

Unfortunately, in conventional "secure" systems, it is often the case that

these environments are not taken into account. Nevertheless, since the environments described above are expected to prevail, it is of critical importance to establish security of a proper level in them.

Motivated by these concerns, we are designing and developing *Saga Security System**, a security architecture in open distributed environments (Information-technology Promotion Agency (IPA), Japan 1996, Soshi 1996, Soshi & Maekawa 1997). In this paper we focus on and discuss its authorization model and mechanism. The authorization model of Saga Security System is called *Saga authorization model*, which provides flexible and uniform protection scheme suitable for object-oriented systems and cooperative agent systems.

The rest of the paper is organized as follows. We describe the overview of Saga Security System in Section 2 and review related work in Section 3. In Section 4, we generally discuss authorization models in open distributed systems. In Section 5, we explore Saga authorization model. We present the security mechanism of Saga Security System in Section 6. Finally we conclude this paper in Section 7.

2 OVERVIEW OF SAGA SECURITY SYSTEM

In open distributed systems, among multiple nodes or applications, information is repeatedly copied or moved as it is. In such environments, once security is compromised in the middle of information flow and unauthorized dissemination of the information takes place, its security can never be controlled anymore. This is one of the most serious security problems inherent in open distributed systems.

Unfortunately, however, the use of mutual authentication protocols and encrypted communications for network security (Popek & Kline 1979) alone is not a satisfactory solution to the problem, because once encrypted information is decrypted, it is not protected anymore. Hence, though the use of encryption technique provides communication security, the ultimate security of information depends on security of the computer that processes the information.

These considerations naturally lead us to the concept of mobile agents or objects (for example, Gosling & McGilton 1995): instead of information being delivered as it is (whether encrypted or not), a dedicated autonomous agent traverses over a network system. Such an agent is a self-contained entity from the viewpoint of security, i.e., it has a security profile for the information, security procedures, audit trail, and so forth. Additionally, information contained in an agent is accessed only through the agent, and the required level of security is attained.

This is the background of *Saga Security System*, a security architecture in open distributed environments. An agent in Saga Security System is called a *Saga Agent*.

---

* We called this system "Security Agent System" before, but we have changed the name because it was too generic and a little confusing.

Henceforth we are mainly concerned with the authorization model and security mechanism of Saga Security System. For further details of Saga Security System, see Information-technology Promotion Agency (IPA), Japan (1996), Soshi (1996), Soshi & Maekawa (1997).

3 RELATED WORK

This section examines the previous work relevant to ours.

In order to take advantage of encapsulation of object-oriented systems, the authorization models where users are given authorization on methods, not on primitive operations, have been an active area of research in recent years (Ahad, Davis, Gower, Lyngback, Marynowski & Onuegbe 1992, Bertino & Samatari 1993). Nevertheless they are not so uniformly modeled as Saga authorization model is. Moreover, surprisingly few models addressed current open distributed environments.

Rabitti, Bertino, Kim & Woelk (1991) developed a sophisticated authorization model on ORION object-oriented database. In the model, exploiting relationships among subjects, objects, and access modes, we can derive implicit authorizations from explicitly specified authorizations. However, the model could not appropriately deal with the situations where agents or objects work in cooperation in distributed systems. In other words, the model is orthogonal to Saga authorization model and it is possible to integrate both of them.

PCM (Path Context Model) (Boshoff & Solms 1989, Olivier & Solms 1992) takes into account potentially insecure distributed environments and can perform access control in terms of access path. Although PCM does not model semantics of authorization, it can express a wide variety of conditions of network environments, for instance, network domains and encrypted channels. It is desirable to accommodate such expressive power into our model in the future.

4 AUTHORIZATION MODEL IN OPEN DISTRIBUTED ENVIRONMENTS

Before we discuss Saga authorization model, let us consider a general authorization model for the moment.

A protection state of a computer system can be represented in an *access control matrix*, A (Maekawa, Oldehoeft & Oldehoeft 1987). Each element of an access control matrix, $A[S, O]$, maintains the set of access rights that a subject S holds on an object O. Here, *objects* are passive entities, e.g., files or I/O devices, protected by security control mechanism, and *subjects* are active entities, e.g., users or processes, that access objects. In order to clarify semantics and terminology, however, in the remainder of the paper, we call a "subject" a "client" and we do not use the generic term "object" but the

specific name of an entity to be accessed. In addition, the term "object" used in the paper is supposed to refer to an "object" in object-oriented systems unless otherwise explicitly stated.

On implementation of an access control matrix, since it is not practical to implement it straightforwardly because of its sparseness, many traditional computer systems implement only either its rows (called *capability*) or columns (called *access control list*, or ACL).

Based on the access control matrix model, in the past decades, considerable amount of research has been done on computer security (Castano, Fugini, Martella & Samarati 1995, Denning 1982). However, substantially little attention has been paid to the security of current open distributed systems. In addition to this, only in the last few years, we have seen studies on security models adequate for advanced application, for example, object-oriented systems and cooperative agent systems.

Therefore, we propose Saga authorization model, which has the following distinguished advantages that are never found in conventional systems:

- The model takes into account open distributed systems.
- The semantics of the model is independent of that of application, so that applications can be developed without constraints of security policies and can be changed without affecting security policies or vice versa.
- The model enables us to specify authorizations appropriate for advanced application such as object-oriented systems and cooperative agent systems.
- Operations of various level of granularity are handled uniformly in the model.
- The model expresses and controls trust relationships in distributed systems.

We examine Saga authorization model in Section 5.

5 SAGA AUTHORIZATION MODEL

Now in this section we discuss Saga authorization model in detail.

5.1 Fine-grained and coarse-grained access control

In traditional authorization models, users can exercise the rights of directly invoking primitive operations, such as read and write, and these models cannot take advantage of the concept of encapsulation in object-oriented systems.

That is exemplified as follows. Suppose that a user can update his own data in a personal information database and is about to change his address field. In conventional approaches, the user is given authorization on primitive write operation for the data. However, since this approach allows the user

to directly access the data, he can update the address in a wrong format or destroy it intentionally or accidentally.

All we have to do in order to counter the problem is as follows: we prepare an operation putAddress, which updates an address field in a prescribed format, and then we give authorization on putAddress, but not on write, to the user. This way we can naturally incorporate encapsulation into access control.

In addition to such coarse-grained access control, fine-grained access control is also significantly useful, for example, where access control is performed together with information flow control (Denning 1982).

From these observations, we see that it is desirable for an authorization model to be able to control *access of various level of granularity*. Therefore in Saga authorization model, operations of various level of granularity can be handled uniformly as *services*.

In the following discussions, $o.s$ stands for a service s implemented in a Saga Agent o when we wish to emphasize who provides a service.

5.2 Access control regarding services as clients

In traditional authorization models, authorization is given to an active entity of a relatively large granularity, such as a user, process, class, and object. However, this approach sometimes fails to provide flexible protection scheme.

To see why, let us consider the following example. Suppose that we have a service getAverageSalary, which computes the average of the salaries of all employees. getAverageSalary invokes a getSalary service to see the salary of each employee. Now, if we grant authorization on a service only to a user, in order to authorize a user u to execute getAverageSalary, we may also have to permit u to execute getSalary for each employee as well. This solution, however, is far from perfect because u can directly invoke getSalary to know the salary of another user.

This problem is easily solved if only we can grant *to a service* the authorization to invoke another service. In the example above, all we have to do is to place the authorization to invoke getSalary on getAverageSalary, not directly on u. This way we can offer fine-grained and flexible protection in Saga authorization model.

5.3 Service context in distributed environments

In centralized systems, when a user invokes a service, the service is executed on behalf of him. However this no longer holds true for distributed systems. That is, in distributed systems, it is often the case that a client resides in one Saga Agent or node but a service invoked by the client resides in another. To control security appropriately for such a situation, we must be able to model trust relationships among cooerative agents in distributed systems. For instance,

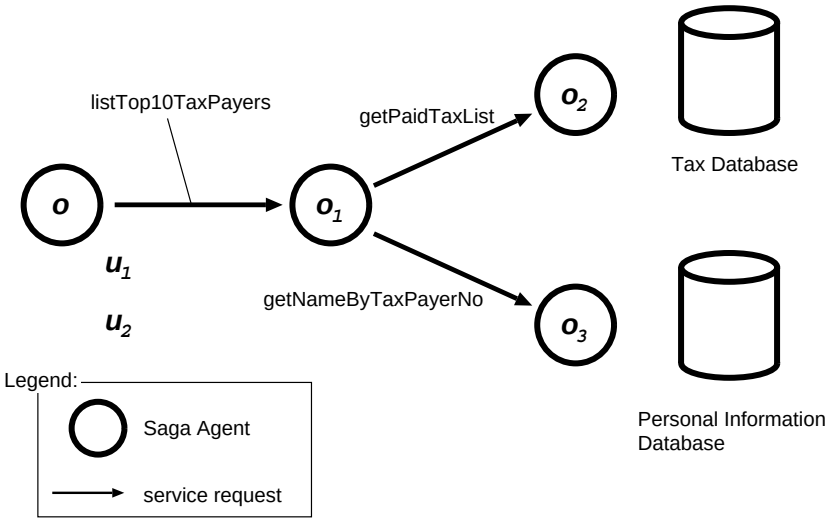

Figure 1 Service Relationships

with respect to a service s of a Saga Agent and users u_1, u_2, and u_3, if we can specify that u_1 is authorized to execute s invoked by u_2 but not s invoked by u_3, we can provide highly flexible protection scheme.

For that purpose, we define the *current user* of a Saga Agent as the user who has invoked the Agent. A Saga Agent executes on behalf of its current user. Furthermore, with respect to a Saga Agent o, a *service context* σ is defined as the pair of a service $o.s$ and its current user u, $(u, o.s)$. In Saga authorization model, authorization is given on the basis of service contexts.

5.4 Service path in distributed environments

In this section, we shall integrate and evolve the designs of Saga authorization model discussed in Section 5.1, 5.2, and 5.3.

When Saga Agents work in cooperation in open distributed environments, it often happens that a service of a Saga Agent invokes a second Saga Agent's service, which further invokes a third one, and so on. Conventional access control matrix models cannot be fully applicable to such situations.

To discuss this, consider the case shown in Figure 1. Saga Agent o_1 provides a service listTop10TaxPayers, which collects data about taxpayers and displays top 10 of them in some expected style. listTop10TaxPayers calls two auxiliary services, namely, getPaidTaxList of a Saga Agent o_2 and getNameByTaxPayerNo of a Saga Agent o_3. getPaidTaxList of o_2 sorts all paid taxes in descending order and returns the result as well as corresponding taxpayer numbers. getNameByTaxPayerNo of o_3 takes a taxpayer number as an argu-

ment and returns the name of the taxpayer. In the following discussion, to avoid unnecessary complexity, we mention only services in service contexts.

Now consider the case where a user u_1 and a user u_2 are about to invoke listTop10TaxPayers. Suppose that u_2 is authorized to know the taxpayer numbers and paid taxes of the returned list, and that u_1 is additionally allowed to know the names of the taxpayers. Moreover, suppose that both u_1 and u_2 are not authorized to directly invoke getPaidTaxList and getNameByTaxPayerNo. Consequently, when u_1 invokes listTop10TaxPayers, both of getPaidTaxList and getNameByTaxPayerNo should be allowed to execute, but when u_2 invokes listTop10TaxPayers, getNameByTaxPayerNo should not be allowed.

At first glance, we could think of the following two solutions to this problem:

1. We grant the authorization to invoke both of getPaidTaxList and getNameByTaxPayerNo to the service listTop10TaxPayers. When the Saga Agent a receives a request for listTop10TaxPayers, o_1 invokes either of or both of getPaidTaxList and getNameByTaxPayerNo in compliance with security policy.
2. According to each user's own authorization, we decompose listTop10TaxPayers into services so that the user may be authorized to invoke some of them.

Nevertheless, these approaches have a fatal drawback that applications must provide protection by themselves because the security models and mechanisms alone cannot deal with the problem above. In other words, applications are imposed restrictions on not only by their own semantics, but also by security policies. Thus, for instance, if u_2 is authorized to call getNameByTaxPayerNo sometime later, o_1 must be reconstructed (possibly from scratch) to incorporate the change in it. To make matters worse, it is nearly impossible that the solutions above apply to more complicated situations, say, getPaidTaxList or getNameByTaxPayerNo further invokes services of other Saga Agents.

Therefore, authorization models in distributed systems must be able to represent, independently of application semantics, a situation where agents or objects work in cooperation as the example above shows. Such situation can be modeled by a path of service contexts invocations, $\sigma_1\sigma_2\ldots\sigma_i$, where σ_1 invokes σ_2, which further invokes σ_3, ..., and finally σ_{i-1} invokes σ_i. Hence in Saga authorization model, we can grant the authorization to invoke a service context σ *to a path of service context invocations*, $\sigma_1\sigma_2\ldots\sigma_i$. That is, a request for σ is authorized if and only if the path of service context invocations resulting in the request exactly matches the sequence $\sigma_1\sigma_2\ldots\sigma_i$.

The concept of service paths is one of the key features of Saga authorization model and is never found in other traditional authorization models. The introduction of service paths makes it possible to uniformly synthesize the designs discussed in Section 5.1, 5.2, and 5.3 into Saga authorization model and to realize flexible and powerful protection in distributed systems.

Now, to demonstrate the effectiveness of the approach, consider again the case depicted in Figure 1. In Saga authorization model, the problem is readily solved as shown in Table 1 ('→' stands for 'invokes' relation). We should note how simply it can be solved and how flexible, uniform, and powerful the our authorization model is.

Table 1 Service Sequences Authorized to Invoke Services

client	*service*	
	getPaidTaxList	getNameByTaxPayerNo
$u_1 \rightarrow$ listTop10TaxPayers	allowed	allowed
$u_2 \rightarrow$ listTop10TaxPayers	allowed	denied

We are now ready to state this idea more formally.

A *service path* is defined as a sequence of service contexts invocations, $\sigma_1\sigma_2 \ldots \sigma_i$. In this paper, a service path is denoted α. Additionally, a *level* is associated with a service path. The level of a service path is defined as i if the service path consists of i service contexts.

In Saga authorization model, a service path is authorized to invoke a service context. To state the situation simply and clearly, we define a *request pair* as the pair of a service path α and a service context σ, (α, σ). Then, when a request routed along α to call σ is allowed, we say that (α, σ) is authorized.

Generally speaking, we can express invocation relationships among services with a tree structure. Hence, with respect to a service path, we use without explanation the terms in graph theory such as parent, children, sibling, ancestor, and descendant.

5.5 Relationships among authorizations

In Saga authorization model, a service path is authorized to invoke a service context. However, direct application of this to real systems is considered impractical due to the following reasons:

- It is extremely hard to correctly specify authorizations for every service context and service path in a system.
- Authorizations of service paths cannot be easily adapted to changes of authorizations or service implementations.
- Relationships among authorizations, service contexts, and service paths are not immediately obvious.

To cope with these problems, we introduce the authorizations of new types, *cover authorization* and *composite authorization*. In addition, to make semantics of authorization clear, when referring to authorization discussed so far, we call it *primitive authorization*. Thus there are three kinds of authorizations we can explicitly grant to a request pair (α, σ), i.e., primitive, cover, or composite authorization, and we call them *explicit authorizations*. Furthermore we assume that at most one of three types of authorizations can be granted to (α, σ).

Now we examine cover authorization and composite authorization.

First we consider the case where *cover authorization* is granted to (α, σ). In this case, (α, σ) is authorized and furthermore for an arbitrary service path α' and service context σ', authorization on a request pair $(\alpha\sigma\alpha', \sigma')$ is derived*. In other words, when cover authorization is specified on (α, σ), all descendant service paths of $\alpha\sigma$ are authorized to invoke any service context. Consequently, cover authorization enables us to easily update services without affecting their ancestor service paths. Moreover, through cover authorization, we can utilize the potential of polymorphism and encapsulation in object-oriented systems.

Next, if *composite authorization* is given to (α, σ), the authorization depends on the authorizations on the children of $\alpha\sigma$. A typical usage of composite authorization on (α, σ) is that (α, σ) is authorized if and only if all children of $\alpha\sigma$ are authorized. To describe composite authorization more formally, first let us define $C(\alpha)$ as a set of service contexts to be invoked by a service path α. We assume that $C(\alpha)$ does not depend on authorizations but on semantics of application. Furthermore, we associate with (α, σ) a logical formula $P_{(\alpha,\sigma)}$ that is combination of conjunction and disjunction of authorization on each request pair $(\alpha\sigma, \sigma_k)$, $k = 1, \ldots, i$, where $\{\sigma_1, \sigma_2, \ldots, \sigma_i\} = C(\alpha\sigma)$. $P_{(\alpha,\sigma)}$ describes relationships between composite authorization on (α, σ) and authorizations on its children. Now, using $P_{(\alpha,\sigma)}$ (and $C(\alpha\sigma)$ implicitly), we can give formal description of composite authorization: if composite authorization is specified on (α, σ), (α, σ) is authorized if and only if $P_{(\alpha,\sigma)}$ is evaluated true.

Cover authorization and composite authorization express relationships among authorizations, simplify the specification of consistent authorizations in a system, and offer flexible authorization scheme.

5.6 Validation of requests

So far we have discussed Saga authorization model thoroughly. In this section, to summarize the discussions, we present access control in Saga authorization model and propose the authorized algorithm, which validates a request pair (α, σ).

To begin with, note that we do not take negative authorization (Rabitti

*Note that authorization is not derived on an arbitrary (α', σ'), but on $(\alpha\sigma\alpha', \sigma')$. That is, cover authorization does not derive excessive authorization at all as it first might appear.

et al. 1991) into account in Saga authorization model. Thus, generally speaking, if explicit authorization is not specified on (α, σ), α is not allowed to execute σ. Only one exception to this is the case in which cover authorization is specified on (α', σ') where $\alpha = \alpha'\sigma'\beta$ holds. In that case, authorization on (α, σ) is derived from the cover authorization.

Next, let us consider the case where primitive authorization or cover authorization is granted to (α, σ). In that case, we can immediately execute σ whether or not authorization on (α, σ) is derived from cover authorization.

Furthermore, we discuss composite authorization. As we have seen in Section 5.5, when composite authorization is given to (α, σ), access control is enforced as follows*: if composite authorization is specified on (α, σ), (α, σ) is authorized if and only if $P_{(\alpha,\sigma)}(\mathsf{authorized}(\alpha\sigma, \sigma_1), \mathsf{authorized}(\alpha\sigma, \sigma_2), \ldots, \mathsf{authorized}(\alpha\sigma, \sigma_i))$ is evaluated true where $C(\alpha\sigma) = \{\sigma_1, \sigma_2, \ldots, \sigma_i\}$.

However, one problem remains in this discussion. That is to say, we encounter a contradiction when $P_{(\alpha,\sigma)}$ is evaluated false meanwhile authorization on (α, σ) is derived from cover authorization. For simplicity and effectiveness, the authorization derived from cover authorization takes precedence over composite authorization in Saga authorization model. Thus (α, σ) is authorized when its authorization is derived from cover authorization, whether or not composite authorization is granted to (α, σ).

Finally, a special case is worth while mentioning – recursive calls of service contexts. For example, it often happens that σ_1 invokes σ_2, which further invokes σ_1, and so forth. Suppose that composite authorization is specified on (α, σ_1) for some α. Since the composite authorization depends on σ_2, at a glance you might think that it would be possible that validation process of (α, σ_1) would end in an infinite loop. However, this is not true since composite authorization is not granted to a service context but to a request pair. Since on each service invocation, the level of the corresponding service path is getting higher, we see that an infinite loop in validation process is never possible. Needless to say, cover authorization is more appropriate for such a case.

Now, from the discussions above, we see that the authorized algorithm, which validates (α, σ), is defined as shown in Figure 2.

6 IMPLEMENTATION

In this section, we present the security mechanism of Saga Security System, highlighting its outstanding features, *access tokens*, *access control vectors*, and *Security Monitors*.

---

*The final form of authorized algorithm is illustrated in Figure 2.

procedure authorized(α, σ)
if primitive authorization or cover authorization is granted to (α, σ)
  then return true
else begin
  if authorization on (α, σ) is derived from cover authorization
    then return true
  else begin
    if composite authorization is granted to (α, σ)
      /* suppose that $C(\alpha\sigma) = \{\sigma_1, \sigma_2, \ldots, \sigma_i\}$ */
      then return $P_{(\alpha,\sigma)}$(authorized$(\alpha\sigma, \sigma_1)$, authorized$(\alpha\sigma, \sigma_2)$,
                    ..., authorized$(\alpha\sigma, \sigma_i)$)
    else return false
  end
end;

Figure 2 authorized algorithm

6.1 Access token

To implement Saga authorization model, we cannot overemphasize the importance of ensuring the authenticity and integrity of a request pair (α, σ), in particular, those of a service path α. These are achieved in Saga Security System with *access tokens*, combined with *public key cryptosystems* (Denning 1982).

In public key cryptosystems, a key for encryption and one for decryption are not identical. An encryption key is called a *public key* and can be made public, on the other hand, a decrptyion key is called a *private key* and must be kept secret. One of the important features of public key cryptosystems is that they can generate a *digital signature* on a message, with which we make sure that the identity and content of the message have not been compromised. In the rest of the paper, K and K^{-1} denote a public key and the corresponding secret key, respectively. In addition, we write $\{M\}_K$ to mean that a message M is encrypted (or decrypted) with a key K.

Using public key technology, Saga Agents submit request messages via *access tokens*. To discuss the structure of an access token, let us consider the case where a Saga Agent o_i sends to a Saga Agent o_{i+1} a request (α_i, σ_{i+1}) where $\alpha_i = \sigma_1\sigma_2 \ldots \sigma_i = (u_1, o_1.s_1)(u_2, o_2.s_2) \ldots (u_i, o_i.s_i)$ and $\sigma_{i+1} = (u_{i+1}, o_{i+1}.s_{i+1})$. τ denotes an access token and let τ_i be the access token corresponding to (α_i, σ_{i+1}). For convenience, τ_0 is supposed to be $(u_1, o_1.s_1)$.

Now τ_i is defined as follows:

$$\tau_i = \{\alpha_i, \sigma_{i+1}, \Pi_i, \Theta_i\} \qquad (i = 1, \ldots)$$

where $\Pi_i = \{I_1, I_2, \ldots, I_i\}$ and I_i includes the constraints specified by o_i, for example, an expiration time of the access token with which o_i can counter

a replay attack (Denning 1982) on it. Furthermore, Θ_i is defined as the set of the signatures, $\{\theta_1, \theta_2, \ldots, \theta_i\}$, where $\theta_i = \{\theta_{i-1}, \sigma_{i+1}, I_i\}_{K_i^{-1}}$. We assume $\theta_0 = \sigma_1 = (u_1, o_1.s_1)$.

The authenticity and integrity of a request pair (α_i, σ_{i+1}) in τ_i is verified in the following manner. Recall that $\alpha_i = \sigma_1 \sigma_2 \ldots \sigma_i$ by definition. Then, receiving τ_i, o_{i+1} can immediately generate i tuples from τ_i, $\{\theta_k, \theta_{k-1}, \sigma_{k+1}, I_k\}$ where $k = 1, \ldots, i$. Next o_{i+1} verifies that $\{\theta_1\}_{K_1} = \{\theta_0, \sigma_2, I_1\} = \{\sigma_1, \sigma_2, I_1\}$ and using I_1, o_{i+1} also verifies that the conditions specified by o_1 are satisfied. If all the verifications succeed, similarly o_{i+1} verifies $\{\theta_2\}_{K_2} = \{\theta_1, \sigma_3, I_2\}$ and the conditions in I_2, ..., and eventually $\{\theta_i\}_{K_i} = \{\theta_{i-1}, \sigma_{i+1}, I_i\}$ and the conditions in I_i. Note that a straightforward implementation of a request pair would be vulnerable to the attack of insertion, deletion, and exchange of service contexts of the service path because a service path is a sequence of service contexts*. However, we have a set of signatures $\theta_k = \{\theta_{k-1}, \sigma_{k+1}, I_k\}_{K_k^{-1}}$, $k = 1, \ldots, i$, in τ_i and using them we provide protection against the attack.

Now, from these discussions, we see that the structure of an access token and a digital signature allow o_i to specify σ_{i+1} and I_1, but not to alter α_i freely. Of course, intruders cannot compromise the authenticity and integrity of access tokens. Therefore, with access tokens, we can realize Saga authorization model in open distributed environments.

6.2 Access control vector

Every Saga Agent has an access control list, ACL, for services implemented in it. An ACL in a Saga Agent is a list of *access control vectors*.

An access control vector is prepared when explicit authorization is granted to a request pair. Since negative authorization is not defined in Saga authorization model, when there does not exist an access control vector corresponding to a request pair, the request pair is not authorized unless authorization on it is derived from cover authorization.

Now let us take a closer look at the structure of an access control vector corresponding to $(\alpha, (u, o.s))$. The access control vector consists of four parts: (1) a service path α, (2) a service s (Saga Agent ID o and current user ID u are not needed here), (3) a reference to the implementation of s, and (4) miscellaneous information, which includes an authorization type and a logical formula P (see Section 5.6).

When an access token for a request $(\alpha, (u, o.s))$ is sent to a Saga Agent, the Saga Agent searches its ACL for an access control vector corresponding to the request. Direct implementation of this search might incur severe performance degradation since we have to compare service contexts of α and those of access control vectors one by one. To avoid this and perform effective search, a Saga

*This kind of vulnerability is similar to that of block cipher(Denning 1982).

Agent sorts its access control vectors in advance and performs binary search over them.

6.3 Access control by security monitor

A service called a *Security Monitor* is associated with every Saga Agent. It is the Security Monitor of a Saga Agent that performs security control over the Saga Agent.

When a Saga Agent o receives an access token for a request (α, σ), its Security Monitor m verifies the token in the way as discussed in Section 6.1. If the test is successful, m validates whether (α, σ) is authorized or not using authorized algorithm. Since o has access control vectors for services implemented in it, the validation can be done locally when primitive authorization or cover authorization is granted to (α, σ).

The question is the case where authorization on (α, σ) is derived from cover authorization and the case where composite authorization is granted to (α, σ). In such cases, if authorization on (α, σ) depends on authorizations on remote Saga Agents' services, m sends query messages to those Saga Agents and collects the responses from them. Based on those responses, m validates (α, σ) and if it is allowed, σ is invoked.

This way, while a Saga Agent travels over an open distributed system, its security can be controlled by the Security Monitor integrated with it.

7 CONCLUSION

In this paper we have discussed so far a new authorization model of Saga Security System, Saga authorization model. The model is uniform and flexible, and is appropriate for advanced computing models such as object-oriented systems and agent systems, in open distributed systems. Additionally, we have presented the security mechanism of Saga Security System, which provides protection of a Saga Agent while it travels over an open distributed system.

Hence, the security model and mechanism of Saga Security System realize a promising security architecture in current open distributed systems, which has been strongly required but cannot be provided by traditional systems.

ACKNOWLEDGMENTS

We would like to thank other members of Saga Security System Project, Norihiko Kameda, Kiyoshi Une, Satoshi Yoshida, Atsuki Tomioka, Keisuke Yamaguchi, and Takeharu Kato for useful discussions and comments.

REFERENCES

Ahad, R., Davis, J., Gower, S., Lyngback, P., Marynowski, A. & Onuegbe, E. (1992), Supporting access control in an object-oriented database language, *in* 'Proc. 3rd International Conference on Extending Database Technology (EDBT)', pp. 184–200.

Bertino, E. & Samatari, P. (1993), Research issues in discretionary authorizations for object bases, *in* 'Proc. OOPSLA'93 Workshop on Security for Object-Oriented Systems', pp. 183–199.

Boshoff, W. H. & Solms, S. H. (1989), 'A path context model for addressing security in potentially non-secure environments', *Computers & Security* **8**(5), 417–425.

Castano, S., Fugini, M., Martella, G. & Samarati, P. (1995), *Database Security*, ACM Press.

Denning, D. E. R. (1982), *Cryptography and Data Security*, Addison-Wesley Publishing Co., Reading, MA.

Gosling, J. & McGilton, H. (1995), The Java language environment: A white paper, Technical report, Sun Microsystems.

Information-technology Promotion Agency (IPA), Japan (1996), Research report on security architecture and automatic user authentication in downsizing environments, Technical Report 159, Information-technology Promotion Agency (IPA), Japan. In Japanese.

Maekawa, M., Oldehoeft, A. E. & Oldehoeft, R. R. (1987), *Operating Systems — Advanced Concepts*, The Benjamin/Cummings Publishing Company, Inc.

Olivier, M. S. & Solms, S. H. (1992), 'Building a secure database using self-protecting objects', *Computers & Security* **11**(3), 259–271.

Popek, G. J. & Kline, C. S. (1979), 'Encryption and secure computer networks', *ACM Computing Surveys* **11**(4), 331–356.

Rabitti, F., Bertino, E., Kim, W. & Woelk, D. (1991), 'A model of authorization for next-generation database systems', *ACM Transactions on Database Systems* **16**(1), 88–131.

Soshi, M. (1996), The design and implementation of the Security Agent mechanism, *in* 'The Collection of Papers in the 15th Technical Presentations', Vol. 15, Information-technology Promotion Agency (IPA), Japan, pp. 225–234.

Soshi, M. & Maekawa, M. (1997), 'The Saga security system–a security architecture for open distributed systems', To appear in the Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems.