Concurrency Control in Multi-Role Association

Tomoya Enokido†
†Rissho University
eno@ris.ac.jp

Makoto Takizawa‡
‡Tokyo Denki University
taki@takilab.k.dendai.ac.jp

A concept of role is widely and significantly used to design and implement a secure information system. In a role-based access control (RBAC) model, a role is modeled to be a set of access rights. A subject doing a job is assigned with a role showing the job function in an enterprise. A subject can play multiple roles in an enterprise. In addition to keeping systems secure, objects have to be consistent in presence of multiple conflicting transactions. Traditional locking protocols and timestamp ordering schedulers are based on principles "first-comer-winner" and "timestamp order" to make multiple conflicting transactions serializable, respectively. In this paper, a transaction with more significant roles is considered to be more significant. If a pair of transactions issue conflicting methods to an object, a method from a more significant transaction is performed before another. We discuss which collection of roles is more significant than others. We discuss a role ordering (RO) scheduler so that multiple conflicting transactions are serializable in a significant dominant relation of roles.

1 Introduction

Information systems like relational database systems [5,8] adopt the role-based access control (RBAC) model [7,9]. A *role* shows a job function like president and secretary in an enterprise. Each person plays one or more than one role. A role is a collection of access rights which a subject who plays the role is allowed to do for objects in an enterprise. An *access right* (or *permission*) is given a pair $\langle o, op \rangle$ of an object o and a method op on the object o. Only if an access right $\langle o, op \rangle$ is granted to a subject s, the subject s is allowed to manipulate the object o through the method op in the basic access control model. Roles, i.e. collections of access rights are granted to a subject in the RBAC model. In the *discretionary* approach [5,8], a subject who is granted a role can further grant the role to another subject.

A *transaction* is an atomic sequence of methods which are performed on objects [1]. A pair of methods *conflict* if and only if (iff) the result obtained by performing the methods depends on the computation order. A pair of transactions are referred to as *conflict* with one another if the transactions manipulate a same object through conflicting methods. A collection of conflicting transactions are required to be serializable in order to keep objects consistent. In order to realize the serializability of multiple conflicting transactions, locking protocols [1] are widely used. A transaction T locks an object before manipulating the object through a method op. Other transactions to manipulate the object in a conflicting manner with the method op have to wait until the transaction T releases the object. Locking protocols are based on a principle that only the first comer is a winner and the others are losers. Another way is a timestamp ordering (TO) scheduler [1]. Each transaction T is stamped time $ts(T)$ when the transaction T is initiated. Transactions are totally ordered in their timestamps. Differently from the locking protocols, objects are manipulated by conflicting transactions in the timestamp order and no deadlock occurs.

The authors [3] define the significancy of roles and discuss the *role ordering* (RO) scheduler where each transaction is associated with only one role. In the RBAC model, each subject is rather granted multiple roles and a transaction issued by the subject is associated with one or more than one roles. We define the significantly precedent relation among collections of multiple roles. Based on the significantly precedent relations, we define which transactions are more significant than others, which manipulate an object in a conflicting manner.

In section 2, we present a system model. In section 3, we define significantly dominant relations among roles. In section 4, we discuss the RO serializability of transactions. In section 5, we discuss the RO scheduler.

2 System Model

2.1 Object-based system

An object-based system is composed of objects [4] distributed in networks. An object is an encapsulation of data and methods for manipulating the data. A method is more abstract than primitive methods like *read* and *write*. A pair of methods op_1 and op_2 *conflict* ($op_1 \diamond op_2$) iff the result obtained by performing the methods depends on the computation order. Otherwise, a pair of the methods op_1 and op_2 are *compatible* ($op_1 \square op_2$).

Multiple transactions are concurrently performed on objects. Multiple conflicting transactions are required to be *serializable* to keep objects mutually consistent [1]. Let T_i be a transaction which issues a method op_{ji} to an object o_j $(i, j = 1, 2)$. Suppose there are a pair of transactions T_1 and T_2 where $op_{11} \diamond op_{12}$ on the object o_1 as well as $op_{21} \diamond op_{22}$ on the object o_2. If op_{11} is performed on o_1 before op_{21}, op_{21} is required to be performed before op_{22} on o_2 according to the serializebility theory [1]. Let $\mathbf{T}$ be a set of transactions $\{T_1, ..., T_n\}$. Let H be a schedule of transactions in $\mathbf{T}$. A transaction T_i *precedes* another transaction T_j ($T_i \rightarrow_H T_j$) in a schedule H iff a method op_i from T_i is performed before a method op_j from T_j and $op_i \diamond op_j$. A schedule H is serializable iff the precedent relation $\rightarrow_H$ is acyclic.

2.2 Roles

In access control models [1,6,7,9], a system is composed of two types of entities, *subject* and *object*. A *role* shows a job function in an enterprise. Each subject s plays a role like *president*. The more significant role a subject which plays, the subject is more significant in an enterprise. More significant subjects should be more prioritized in transaction processing than less significant subjects. If a pair of tasks in different jobs use an object, one task in a more significant job should take the

object earlier than the other. A task is realized as a transaction.

A role is a collection of *access rights* in the RBAC model [7]. An access right is specified in a pair $\langle o, op \rangle$ of an object o and a method op. An access request to manipulate an object o by a method op is written in a same form $\langle o, op \rangle$ as an access right. A subject s is first granted a role R. Then, the subject can issue an access request op to an object o only if an access right $\langle o, op \rangle$ is included in the role R. A subject s can be granted multiple roles R_1, ..., R_m. Let $RS(s)$ be a family $\{R_1, ..., R_m\}$ of roles granted to a subject s. A transaction T_i issued by a subject s is associated with a subset of the roles granted to the subject s. Let $subject(T_i)$ denote a subject which initiates a transaction T_i. Thus, each transaction T_i is associated with multiple roles R_{i1}, ..., R_{im_i}. Let $RT(T_i)$ show a family $\{R_{i1}, ..., R_{im_i}\}$ of roles which are associated to a transaction T_i, $RS(T_i) \subseteq RS(subject(T_i))$. A role R_1 is referred to as *inherit* another role R_2 iff $R_2 \subseteq R_1$. Let AC be a set of possible access rights in a system. A power set 2^{AC} gives a collection of possible roles.

3 Significancy on Roles

3.1 Significancy of methods

There are two types of methods, *class* and *object* methods for a class. Class methods are ones for *creating* and *dropping* an object for the class. On the other hand, object methods are ones for manipulating an object of the class. There are two types of object methods, *change* and *output* types. An *output* type of method is a method for deriving data from an object. A *change* type of method is one for changing a state of an object. Some method can be both *output* and *change* types.

Let us consider a pair of methods *withdraw* and *deposit* on a *bank* object. Both the methods *withdraw* and *deposit* are a change type. In our life, a subject *person* more carefully issues a method *withdraw* than a method *deposit* because the account value in the *bank* object is decremented through *withdraw*. That is, some methods are considered to be more significant than others in an application. A method *withdraw* is referred to as *semantically significantly dominate* a method *deposit* (*withdraw* $\succcurlyeq$ *deposit*) on the *bank* object. For a pair of class methods *create* and *drop*, we cannot decide which one is more significant. Here, *create* and *drop* are referred to as *semantically equivalent* (*create* $\cong$ *drop*). A method op_1 is referred to as *semantically significantly equivalent* with another method op_2 ($op_1 \cong op_2$) if $op_1 \succcurlyeq op_2$ and $op_2 \succcurlyeq op_1$. op_1 is *more semantically significant* than op_2 ($op_1 \succ op_2$) if $op_1 \succcurlyeq op_2$ and $op_1 \ncong op_2$. A pair of methods op_1 and op_2 are *semantically uncomparable* on an object o ($op_1 \mid op_2$) iff neither $op_1 \succcurlyeq op_2$ nor $op_2 \succcurlyeq op_1$.

[Definition] A method op_1 is *more significant* than another method op_2 on an object o ($op_1 \succ op_2$) iff one of the following conditions is satisfied:

1. op_1 is a class type and op_2 is an object type.
2. op_1 and op_2 are an object type where op_1 is a *change* type and op_2 is just an *output* one.
3. Object types of methods op_1 and op_2 are same types and $op_1 \succ op_2$.

A method op_1 is *significantly equivalent* with another method op_2 on an object o ($op_1 \equiv op_2$) iff op_1 and op_2 are a same type and $op_1 \cong op_2$. op_1 *significantly dominates* op_2 on an object o ($op_1 \succeq op_2$) iff $op_1 \succ op_2$ or $op_1 \equiv op_2$. A pair of methods op_1 and op_2 are *significantly uncomparable* on an object o ($op_1 \parallel op_2$) iff neither $op_1 \succeq op_2$ nor $op_2 \succeq op_1$.

3.2 Significancy of access rights

We discuss which access right $\langle o_1, op_1 \rangle$ or $\langle o_2, op_2 \rangle$ is more significant than the other based on the significantly dominant relation $\succeq$ of methods. First, objects are classified into some security classes [2]. An object o_1 is *more significant* than another object o_2 ($o_1 \succ o_2$) if o_1 is more secure than o_2 in an enterprise. A pair of objects o_1 and o_2 are *significantly equivalent* ($o_1 \equiv o_2$) if o_1 and o_2 are classified into a same security class. o_1 *significantly dominates* o_2 ($o_1 \succeq o_2$) iff $o_1 \succ o_2$ or $o_1 \equiv o_2$. A pair of objects o_1 and o_2 are *significantly uncomparable* ($o_1 \parallel o_2$) iff neither $o_1 \succeq o_2$ nor $o_2 \succeq o_1$.

[Definition] An access right $\langle o_1, op_1 \rangle$ is *more significant* than another access right $\langle o_2, op_2 \rangle$ ($\langle o_1, op_1 \rangle \succ \langle o_2, op_2 \rangle$) iff 1) $o_1 \succ o_2$ or 2) $op_1 \succ op_2$ if $o_1 \equiv o_2$.

A pair of access rights $\langle o_1, op_1 \rangle$ and $\langle o_2, op_2 \rangle$ are *significantly equivalent* ($\langle o_1, op_1 \rangle \equiv \langle o_2, op_2 \rangle$) iff 1) $op_1 \equiv op_2$ if $o_1 = o_2$, 2) else $o_1 \equiv o_2$. A pair of access rights $\langle o_1, op_1 \rangle$ *significantly dominates* another access right $\langle o_2, op_2 \rangle$ ($\langle o_1, op_1 \rangle \succeq \langle o_2, op_2 \rangle$) iff $\langle o_1, op_1 \rangle \succ \langle o_2, op_2 \rangle$ or $\langle o_1, op_1 \rangle \equiv \langle o_2, op_2 \rangle$. A pair of access rights $\langle o_1, op_1 \rangle$ and $\langle o_2, op_2 \rangle$ are *significantly uncomparable* ($\langle o_1, op_1 \rangle \parallel \langle o_2, op_2 \rangle$) iff neither $\langle o_1, op_1 \rangle \succeq \langle o_2, op_2 \rangle$ nor $\langle o_2, op_2 \rangle \succeq \langle o_1, op_1 \rangle$. Here, $\succeq$ is transitive.

3.3 Significancy of roles

We discuss which role is more significant than another role based on the *significantly dominant* relation $\succeq$ of access rights.

[Definition] A role R_1 *significantly dominates* another role R_2 ($R_1 \succeq R_2$) if for every access right $\langle o_2, op_2 \rangle$ in R_2, there is at least one access right $\langle o_1, op_1 \rangle$ in R_1 such that $\langle o_1, op_1 \rangle \succeq \langle o_2, op_2 \rangle$ and no access right $\langle o_3, op_3 \rangle$ in R_2 such that $\langle o_3, op_3 \rangle \succeq \langle o_1, op_1 \rangle$.

A role R_1 is *significantly equivalent* with another role R_2 ($R_1 \equiv R_2$) if $R_1 \succeq R_2$ and $R_2 \succeq R_1$. A pair of roles R_1 and R_2 are *significantly uncomparable* ($R_1 \parallel R_2$) iff neither $R_1 \succeq R_2$ nor $R_2 \succeq R_1$. A least upper bound (*lub*) $R_1 \sqcup R_2$ of roles R_1 and R_2 is a role R_3 such that $R_3 \succeq R_1$ and $R_3 \succeq R_2$ and there is no role R_4 such that $R_3 \succeq R_4 \succeq R_1$ and $R_3 \succeq R_4 \succeq R_2$. A greatest lower bound (*glb*) $R_1 \sqcap R_2$ is similarly defined. On the other hand, $R_1 \cup R_2$ and $R_1 \cap R_2$ show the union and intersection of roles R_1 and R_2, respectively. $R_1 \cup R_2$ includes every access right in R_1 or R_2. $R_1 \cap R_2$ is composed of access rights in both R_1 and R_2. Let $R_1 + R_2$ denote a family of roles R_1 and R_2.

Suppose a subject s is granted roles R_1, ..., R_m ($m \geq 1$). Here, let $RS(s)$ be a family $R_1 + \cdots + R_m$ of the roles granted to a subject s. Let $AS(s)$ denote a set of access rights granted to a subject s with roles, $AS(s) = \cup_{R \in RS(s)} R$. A transaction T issued by a subject s is associated with a subset of the roles granted to the subject s. $R_1 \sqcup \cdots \sqcup R_m$ and $R_1 \sqcap \cdots \sqcap R_m$ show a least upper bound (*lub*) and a greatest lower bound (*glb*) of the roles R_1, ..., R_m, respectively. Here, $R_1 \sqcap$

$\cdots \sqcap R_m \preceq R_i \preceq R_1 \sqcup \cdots \sqcup R_m$ for every role R_i (i = 1, ..., m). $R_1 \sqcup \cdots \sqcup R_m$ significantly dominates every role R_i ($R_1 \sqcup \cdots \sqcup R_m \succeq R_i$). However, $R_i \preceq R_1 \cup \cdots \cup R_m$ may not hold. Let $RL(T)$ and $RG(T)$ denote the *lub* $\sqcup_{R \in role(T)} R$ and the *glb* $\sqcap_{R \in role(T)} R$ of roles associated to a transaction T, respectively. Let $RU(T)$ and $RI(T)$ show the union $\cup_{R \in role(T)} R$ and the intersection $\cap_{R \in role(T)} R$, respectively.

[Definition] Let $\mathbf{R}_i$ and $\mathbf{R}_j$ be families of roles $\{R_{i1}$, ..., $R_{im_i}\}$ and $\{R_{j1}$, ..., $R_{jm_j}\}$, respectively. $\mathbf{R}_i$ *significantly dominates* $\mathbf{R}_j$ ($\mathbf{R}_i \succeq \mathbf{R}_j$) iff $R_{i1} \sqcap \cdots \sqcap R_{im_i} \succeq R_{i1} \sqcup \cdots \sqcup R_{jm_j}$.

[Definition] A transaction T_i *significantly dominates* a transaction T_j (T_i *S-dominates* ($\succeq$) T_j) iff $RG(T_i) \succeq RL(T_j)$.

Suppose each transaction T_i is associated with roles R_{i1}, ..., R_{im_i} ($m_i \geq 1$, i = 1, ..., n). A transaction T_i issues methods to objects. Here, the transaction T_i can issue only a subset of the access requests in the roles. We consider a pair of cases that every object to be manipulated by a transaction is *a priori* known and every access request to be issued by a transaction is *a priori* known before the transaction is performed. In the first case, it might be unknown what methods to be issued even if it is known what objects to be manipulated by a transaction. Here, let $Ac(T_i)$ be a set of access requests to be issued by a transaction T_i and $Ob(T_i)$ be a set of objects to be manipulated by T_i. Let COM_{ij} be a set of common objects which are manipulated by a pair of transactions T_i and T_j, i.e. $COM_{ij} = Ob(T_i) \cap Ob(T_j)$. Let $role_j(T_i)$ be a family of roles which includes an access right on an object in COM_{ij}, $role_j(T_i) = \{R \mid R \in role(T_i), \exists \langle o, op \rangle \in R, o \in COM_{ij}\}$, $role_j(T_i) \subseteq role(T_i)$. We consider only roles on objects which are to be manipulated by the transactions T_i and T_j.

[Definition] A transaction T_i *significantly dominates* a transaction T_j with respect to objects (T_i *OS-dominates* ($\underline{\succeq}$) T_j) iff $\sqcap_{R_i \in role_j(T_i)} R_i \succeq \sqcup_{R_j \in role_i(T_j)} R_j$.

[Theorem] For every pair of transactions T_i and T_j, $T_i \succeq T_j$ if $T_i \underline{\succeq} T_j$.

If types of methods to be issued by each transaction is *a priori* known, we can reduce the number of roles to be considered. Let $Arole_j(T_i)$ be a subset of $role_j(T_i)$ which include an access request to be issued by a transaction T_i in $Ac(T_i)$, $Arole_j(T_i) = \{R \mid R \in role(T_i)$ and $\exists \langle o, op \rangle \in R$, T_i issues $\langle o, op \rangle\}$, $Arole_j(T_i) \subseteq role_j(T_i)$.

[Definition] A transaction T_i *significantly dominates* a transaction T_j with respect to access rights (T_i *AS-dominates* ($\underline{\underline{\succeq}}$) T_j) iff $\sqcap_{R_j \in Arole_j(T_i)} R_i \succeq \sqcup_{R_j \in Arole_i(T_j)} R_j$.

[Theorem] For every pair of transactions T_i and T_j, $T_i \underline{\succeq} T_j$ if $T_i \underline{\underline{\succeq}} T_j$.

It depends on how well each transaction is *a priori* defined which type of significant dominant relation can be used. If every transaction is the greatest well defined, every access request to be issued is well known before the transaction is performed. Here, the *AS-dominant* relation can be used. If only objects to be manipulated is known before the transaction is performed, the *OS-dominant* relation can be used. Otherwise, just the *S-dominant* relation $\succeq$ can be used.

3.4 Significancy of subjects

A transaction T issued by a subject s is associated with roles granted to the subject s, $role(s)$. We take the discretionary approach [5,8] to adopting the RBAC model [7] to object-based systems. Let $\mathbf{S}(R)$ be a set of subjects which are granted a role R. Subjects in the set $\mathbf{S}(R)$ are partially ordered in the significantly precedent relation $\succ_R$. We define a significant relation among subjects assigned multiple roles. Suppose a pair of subjects s_i and s_j are granted roles R_{i1}, ..., R_{im_i} and R_{j1}, ..., R_{jm_j}, respectively, $role(s_i) = \{R_{i1}$, ..., $R_{im_i}\}$ and $role(s_j) = \{R_{j1}$, ..., $R_{jm_j}\}$ (m_i, $m_j \geq 1$).

[Definition] A subject s_i *significantly* (*S-*)*dominates* another subject s_j ($s_i \succeq s_j$) iff $R_{i1} \sqcup \cdots \sqcup R_{im_i} \succeq R_{j1} \sqcup \cdots \sqcup R_{jm_j}$.

A pair of subjects s_i and s_j are *significantly uncomparable* ($s_i \parallel s_j$) iff neither $s_i \succeq s_j$ nor $s_j \succeq s_i$. A pair of subjects s_i and s_j are *significantly equivalent* with one another ($s_i \equiv s_j$) if $s_i \succeq s_j$ and $s_j \succeq s_i$. If a pair of subjects s_i and s_j are significantly equivalent ($s_i \equiv s_j$), the subjects are ordered based on the *grant* relation as follows: s_i significantly dominates s_j ($s_i \succeq s_j$) if $s_i \succ_R s_j$ for every role R in $role(s_i) \cap role(s_j)$.

3.5 Significancy of transactions

Transactions are ordered in the significantly precedent relation of roles and subjects. Let $\rhd$ show a type of significant dominant relation, $\rhd \in \{\succeq, \underline{\succeq}, \underline{\underline{\succeq}}\}$. For example, if every transaction is well defined in a system, we can take the *AS-dominant* relation $\underline{\underline{\succeq}}$ as $\rhd$.

[Definition] A transaction T_i *significantly precedes* another transaction T_j ($T_i \approx\!\!\succ T_j$) iff $T_i \rhd T_j$ or $subject(T_i) \succeq subject(T_j)$ if T_i and T_j are equivalent with respect to the significant relation $\rhd$.

A pair of transactions T_i and T_j are *significantly equivalent* ($T_i \equiv T_j$) if neither $T_i \succeq T_j$ nor $T_j \succeq T_i$.

4 Serializability

Let $\mathbf{T}$ be a set of transactions which are being performed in a system. A least upper bound (*lub*) $T_1 \cup T_2$ of transactions T_1 and T_2 is a transaction T_3 where $T_3 \approx\!\!\succ T_1$ and $T_3 \approx\!\!\succ T_2$ and there is no transaction T_4 such that $T_3 \approx\!\!\succ T_4 \approx\!\!\succ T_1$ and $T_3 \approx\!\!\succ T_4 \approx\!\!\succ T_2$. A greatest lower bound (*glb*) $T_1 \cap T_2$ is defined similarly. We assume that a top transaction $\top$ and a bottom transaction $\bot$ exist where $\top \approx\!\!\succ T \approx\!\!\succ \bot$ for every transaction T.

A *schedule* H is an execution sequence of methods from transactions in $\mathbf{T}$. A transaction T_1 *precedes* another transaction T_2 in H ($T_1 \rightarrow_H T_2$) iff a method op_1 from T_1 is performed before a method op_2 from T_2 which conflicts with op_1. A schedule H is *serializable* iff the precedent relation $\rightarrow_H$ is acyclic according to the traditional theory [1]. A schedule H of a transaction set $\mathbf{T}$ is shown in a partially ordered set $\langle \mathbf{T}, \rightarrow_H \rangle$.

[Definition] A transaction T_1 *significantly precedes* another transaction T_2 in a schedule H of a transaction set $\mathbf{T}$ ($T_1 \Rightarrow_H T_2$) iff $T_1 \rightarrow_H T_2$ and $T_1 \approx\!\!\succ T_2$, i.e. $T_1 \Rightarrow_H T_2$ if $op_1 \diamond op_2$ and op_1 is performed before op_2 for every pair of op_1 and op_2 from T_1 and T_2, respectively.

Suppose $T_1 \rightarrow_H T_2$. Here, if $T_1 \approx\!\!\succ T_2$, a precedent relation "$T_1 \rightarrow_H T_2$" is *legal* in H. If $T_2 \approx\!\!\succ T_1$, "$T_1 \rightarrow_H T_2$" is *illegal* in H. A schedule $H = \langle \mathbf{T}, \rightarrow_H \rangle$ is *legal* iff $T_1 \rightarrow_H T_2$ if $T_1 \approx\!\!\succ T_2$ for every pair of transactions

T_1 and T_2 in H in $\mathbf{T}$. This means, for every pair of conflicting methods op_1 and op_2 from transactions T_1 and T_2 where $T_1 \approx\!\!> T_2$, op_1 is performed before op_2.

A schedule H $\langle \mathbf{T}, \rightarrow_H \rangle$ is partitioned into subschedules H_1, ..., H_m where each subschedule $H_i = \langle \mathbf{T}_i, \rightarrow_{H_i} \rangle$ $(i = 1, ..., m)$ satisfies the following conditions:

[Role ordering (RO) partition conditions]

1. $\mathbf{T}_i \cap \mathbf{T}_j = \phi$ for every pair of subschedules H_i and H_j and $\mathbf{T}_1 \cup \cdots \cup \mathbf{T}_n = \mathbf{T}$.
2. A precedent relation $T_1 \rightarrow_H T_2$ is legal in H if $T_1 \rightarrow_{H_i} T_2$ for every pair of T_1 and T_2 in $\mathbf{T}_i$ of H_i.
3. For every pair of subschedules H_i and H_j, if $T_{i1} \rightarrow_H T_{j1}$ for some pair of transactions T_{i1} in H_i and T_{j1} in H_j, there are no pair of transactions T_{i2} in H_i and T_{j2} in H_j such that $T_{j2} \rightarrow_H T_{i2}$.

[Definition] A history H of $\mathbf{T}$ is *RO serializable* iff the schedule H is RO partitioned.

It is straightforward for the following theorem to hold.

[Theorem] A history H is serializable if H is RO serializable.

5 Role-Ordering (RO) Scheduler

5.1 One-object model

We discuss an RO scheduler for a single object. Multiple transactions issue methods to an object o. A transaction lastly issues a *commit* (c) or *abort* (a) method. An RO scheduler is composed of a receipt queue RQ and an auxiliary receipt queue ARQ. Let $Tr(op)$ show a transaction which issues a method op. The following procedures are supported to manipulate a queue Q:

1. **enqueue**(op, Q) : op is enqueued into Q.
2. op := **dequeue**(Q) : op is dequeued from Q.
3. op := **top**(Q) : op is a top method in Q.
4. **ROsort**(Q) : all methods in Q are sorted in the significantly precedent relation $\approx\!\!>$ of transactions.

Variables $\mathbf{E}$ and $\mathbf{TE}$ show sets of methods and transactions being currently performed on an object o, respectively. A variable $\mathbf{C}$ denotes a transaction which is performed on the object o and which is significantly preceded by every transaction performed. Initially, $\mathbf{C} := \top$. There are following procedures to perform a method op on the object o:

1. **conflict**(op, $\mathbf{E}$) : **false** if $\mathbf{E} = \phi$ or a method op does not conflict with every method in $\mathbf{E}$, else **true**.
2. **perform**(op) : a method op is performed on the object o.

Suppose methods in transactions T_1, ..., T_m are being performed, $\mathbf{TE} = \{T_1, ..., T_m\}$. Methods of the transactions T_1, ..., T_m being performed are stored in the variable $\mathbf{E}$. Here, a variable $\mathbf{C}$ shows a transaction T_i where $T_j \approx\!\!> T_i$ for every $j = 1, ..., m$. If $\mathbf{C} \approx\!\!> T$, the method op is enqueued into RQ. However, if $T \approx\!\!> \mathbf{C}$, the method op is enqueued into ARQ.

[Delivery of a method op from a transaction T]

```
if T ∈ TE or C ≈> T {
    enqueue(op, RQ); ROsort(RQ);
} else { C := ⊥;  enqueue(op, ARQ); }
```

Methods in the receipt queue RQ are performed on an object o as follows:

[Execution of methods]

```
1. if TE = φ, {
     C := ⊤;
     Every method op in ARQ is moved to RQ;
     ROsort(RQ); /*one subschedule is ended and
      a new schedule is started.*/ }
2. op = top(RQ);
3. if conflict(op, E), return;
   else{ op : = dequeue(RQ);
         if Tr(op) ∉ TE, TE := TE ∪ {Tr(op)};
         E : = E ∪ {op};
         if C ≈> Tr(op), C := Tr(op);
         perform(op); }
```

If op completes, the following procedure is performed:

[Completion of method op]

1. $\mathbf{E} : = \mathbf{E}$ - $\{op\}$;
2. $\mathbf{TE}$:= $\mathbf{TE}$ - $\{Tr(op)\}$ **if** $op = c$ **or** $op = a$;
3. Methods in RQ are performed in the **execution** procedure presented here.

If a top method op_1 conflicting with some method being performed is kept waiting in the receipt queue RQ, every other method in RQ is required to be waited. We discuss how to improve the performance.

[Definition] A method op is *ready* in RQ iff op is compatible with not only every method in being performed but also every waiting method preceding op in RQ.

We introduce the following procedures:

1. **ready**(op, RQ, $\mathbf{E}$) : ***true*** if a method op is ready in RQ, else ***false***.
2. op_1: = **next**(op, RQ) : op_1 is a method in RQ which directly follows an method op.

Let op be a top method in the receipt queue RQ. If op conflicts with some method being performed, the following procedure is performed:

```
op : = top(RQ);
  if conflict(op, E), {
    op : = next(op, RQ);
    while(op ≠ NULL) {
      if ready(op, RQ, E), {
        op is removed from RQ; E : = E ∪ {op};
        TE : = TE ∪ {Tr(op)} if Tr(op) ∉ TE;
        if C ≈> Tr(op), C := Tr(op);
        perform(op); break;
      } else op : = next(op, RQ);
  }}
```

[Theorem] A schedule of a transaction set $\mathbf{T}$ obtained by the RO scheduler is RO-serializable.

[Proof] A subschedule obtained from the receipt queue RQ is RO subschedule. A schedule of the transaction set $\mathbf{T}$ is RO partitioned into the subsequences.

5.2 Distributed object model

In a distributed model, there are multiple objects o_1, ..., o_l $(l > 1)$ distributed in servers and multiple transactions T_1, ..., T_m $(m > 1)$ on multiple clients c_1 ..., c_n $(n > 1)$. Let $mset(T_t)$ be a set of requests which will be issued by a transaction T_t and $oset(T_t)$ be a set of objects to be manipulated, $\{o \mid \langle o, op \rangle \in mset(T_t)\}$ $(t = 1, ..., m)$. Each transaction T_t first sends $mset(T_t)$ to every object o_i to be manipulated in $oset(T_t)$. After sending $mset(T_t)$, the transaction T_t issues methods to the objects. A transaction T_t lastly issues a *commit* (c) or an *abort* (a) method to every object in $oset(T_t)$.

Each client c_s has a sequence number f which is initially one ($f = 1$) ($s = 1, ..., n$). Each client c_s periodically sends a *fence* message k which includes the variable kf which shows the sequence number f of the client. After sending the *fence* message, the client c_s increments the variable f by one.

There are local receipt queues $RQ_{i1}, ..., RQ_{in}$ in each object o_i ($i = 1, ..., l$). Methods and $mset(T_t)$ issued from each transaction T_t on a client c_s to an object o_i are stored in each local receipt queue RQ_{is} ($s = 1, ..., n$). We assume a communication network supports every pair of an object o_i and a client c_s with a reliable communication channel. Requests in local receipt queues RQ_{i1}, ..., RQ_{in} are moved to a global receipt queue GRQ_i on the object o_i. Here, requests in GRQ_i are sorted in the significantly precedent relation $\approx\!\!>$ of transactions. The following conditions have to be satisfied for a collection of GRQ_1, ..., GRQ_l for objects o_1, ..., o_l, respectively, to realize the serializability of multiple transactions:

[Role-based serializability (RBS) conditions]

1. Methods in every global receipt queue GRQ_i are sorted in the significantly precedent relation $\approx\!\!>$ of transactions ($i = 1, ..., l$).
2. For a top method op_t from a transaction T_t in each global receipt queue GRQ_i, if there is a method op_u from a transaction T_u in GRQ_i which op_t precedes and conflicts with op_u, op'_t precedes op'_u in every GRQ_j where op'_t and op'_u are methods form T_t and T_u, respectively, and $op'_t \diamond op'_u$.

We discuss how the RO scheduler on each object handles methods and $mset(T_l)$ received from multiple transactions on multiple clients in order to satisfy the RBS conditions. Each object o_i has a subsequence number variable f_{o_i} which initialized to be one ($f_{o_i} = 1$).

[Receiving procedure]

If there is a *fence* message k_s where $k_s \cdot f = f_{o_i}$ in every local receipt queue RQ_{is} of an object o_i, every request r preceding the *fence* message k_s is dequeued from every local receipt queue RQ_{is}. Then, one of the following procedures is performed for r.

1. If $r = mset(T_t)$, $mset(T_t)$ is stored into an auxiliary global receipt queue ($AGRQ_i$) of the object o_i so that request messages are sorted in $\approx\!\!>$. If there is already a fence message k in $AGRQ_i$, the $mset(T_t)$ is stored only after the *fence* message k.
2. If the request message r is a method op_{it} issued by a transaction T_t to the object o_i ($r = op_{it}$), one of the following procedures is performed in the RO scheduler:
   - 2.1 If $mset(T_t)$ is stored between the top of $AGRQ_i$ and a *fence* message k, op_{it} is stored between the top of GRQ_i and k of the GRQ_i so that request messages are sorted in $\approx\!\!>$.
   - 2.2 If $mset(T_t)$ is stored between a *fence* message k and another *fence* message k' in $AGRQ_i$, op_{it} is stored between k and k' of GRQ_i so that request messages are sorted in $\approx\!\!>$.
   - 2.3 If $mset(T_t)$ is stored between a *fence* message k and the end of $AGRQ_i$, op_{it} is stored between k and the end of GRQ_i so that request messages are sorted in $\approx\!\!>$.
3. If all top messages in all the local receipt queues RQ_{i1}, ..., RQ_{in} are *fence* messages k_s where $k_s \cdot f = f_{o_i}$, the RO scheduler enqueues a *fence* message k where $kf = f_{o_i}$ into GRQ_i and $AGRQ_i$. After that, the RO scheduler removes k_s in the top of every RQ_{is} and increments the variable f_{o_i} by one.

Next, we discuss how the RO scheduler on an object o_i delivers methods from GRQ_i to an object o_i.

[Delivery procedure]

1. If the top method of GRQ_i is a method op_{it} and the following two conditions are satisfied, the method op_{it} is delivered to an object o_i.
   - 1.1 op_{it} does not conflict with every method currently performed on the object o_i.
   - 1.2 Transactions which conflict with the transaction T_t and precede the transaction T_t in $AGRQ_i$ are completed on the object o_i.
2. If the top method of GRQ_i is a *fence* message k, the RO scheduler waits for completion of every method being currently performed on the object o_i. After every method being currently performed is completed, the *fence* message k is removed from GRQ_i.

If a method op completes, the following procedure is performed.

[Completion of a method]

1. If the method op is *commit* (c) or *abort* (a) of a transaction T_t, the RO scheduler removes $mset(T_t)$ of the transaction T_t from $AGRQ_i$.

6 Concluding Remarks

We discussed a role ordering (RO) scheduler based on role concept. The role is a central concept to design, implement, and operate information systems. In this paper, multiple conflicting transactions are serializable according to the significantly dominant relation of roles. We also discussed the RO scheduler for single-server and multi-server models and how to implement the RO scheduler.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. *Addison-Wesley*, 1987.

[2] D. E. Denning and P. J. Denning. *Cryptography and Data Security*. Addison-Wesley Publishing Company, 1982.

[3] T. Enokido and M. Takizawa. Concurrency Control using Role Ordering (RO) Scheduler. *Proc. of the IEEE 19th International Conference on Advanced Information Networking and Applications (AINA-2005)*, Vol. 1:755–760, 2005.

[4] O. M. G. Inc. The Common Object Request Broker : Architecture and Specification. *Rev. 2.1*, 1997.

[5] Oracle Corporation. Oracle8i Concepts Vol. 1. 1999. Release 8.1.5.

[6] R. S. Sandhu. Lattice-Based Access Control Models. *IEEE Computer*, Vol. 26(No. 11):9–19, 1993.

[7] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, Vol. 29(No. 2):38–47, 1996.

[8] Sybase. Sybase SQL Server. http://www.sybase.com/.

[9] Z. Tari and S. W. Chan. A Role-Based Access Control for Intranet Security. *IEEE Internet Computing*, Vol. 1:24–34, 1997.