Concurrency Control for Distributed Objects using Role Ordering (RO) Scheduler

Tomoya Enokido†, Runhe Huang‡, and Makoto Takizawa†
†Tokyo Denki University
E-mail {eno, taki}@takilab.k.dendai.ac.jp
‡Hosei University
E-mail rhuang@k.hosei.ac.jp

A concept of role is significant to design and implement a secure information system. A role shows a job function in an enterprise. In addition to keeping systems secure, objects have to be consistent in presence of multiple 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. We define a significantly precedent lation on roles showing which one of a pair of roles is more significant than another one in an enterprise. We discuss a scheduler so that multiple conflicting transactions are serializable in a significant order of roles of transactions.

ロール順序付け (RO) スケジューラを用いた同時実行制御

榎戸 智也†, 黄 潤和‡, 滝沢 誠†
† 東京電機大学理工学部情報システム工学科
‡ 法政大学情報科学部

役割 (role) は対象世界 (企業等) の業務に対応し, アクセス権限の集合として定義される. システム内で複数のトランザクションが並列に実行されるとき, システム内のオブジェクトを正しい状態に保つためには, 競合する複数のトランザクションを直列に実行する必要がある. 競合する複数のトランザクションを直列化する方法として, ロック, 時刻印順序付けプロトコルが提案されている. これらの手法は「早いもの勝ち」および「時刻印順」でトランザクションを直列化している. 役割は, 対象世界の業務に対応することから重要度の高い業務は優先して行われるべきである. 本論文では, 複数のトランザクションを役割の重要度を基に直列化して実行する新たな同時実行制御方法を提案する。

1 Introduction

Information systems like relational database systems [5,7] adopt role-based access control (RBAC) models [6,8]. A *role* shows a job function like president and secretary, which each person performs in an enterprise. A role is a collection of access rights which a subject who plays the role is allowed to do for objects in an enterprise. Here, an *access right* (or *permission*) is 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 *discretionary* approach [5, 7], 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 [1,3]. A pair of methods *conflict* if and only if (iff) the result obtained by performing the methods depends on the computation order. Transactions are referred to as *conflict* 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, 3] are widely used. A transaction T locks an object before manipulating the object by 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 when the transaction T is initiated, timestamp $ts(T)$. 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.

In this paper, we discuss a concurrency control algorithm based on roles associated for transactions, *role ordering* (*RO*) scheduler. Let T_1 and T_2 be a pair of transactions which are associated with roles R_1 and R_2, respectively, and which manipulate an object o in a conflicting manner. Here, the transaction T_1 manipulates the object o before T_2 if the role R_1 is more *significant* than the other role R_2. This means the more significant job a transaction does, the earlier an object can be manipulated by the transaction. In the RO scheduler, conflicting methods issued by transactions are ordered in the significancy of the roles. Transactions can concurrently manipulate objects in such an order that persons really do their jobs in an enterprise.

In section 2, we present a system model. In section 3, we newly define significantly dominant relations among roles. In section 4, we discuss the role ordering (RO) serializability. In section 5, we discuss the role ordering (RO) scheduler. In section 6, we evaluate the RO scheduler compared with the two-phase locking (2PL) protocol.

2 System Model

2.1 Object-based system

A system is composed of objects [4] which are distributed on multiple computers in networks. An object is an encapsulation of data and methods for manipulating the data. An object can be manipulated only through methods. A method is more abstract than primitive methods like *read* and *write*. A pair of methods op_1 and op_2 supported by an object o are referred to as *conflict* with one another iff the result obtained by performing the methods op_1 and op_2 depends on the computation order. Otherwise, a pair of the methods op_1 and op_2 are *compatible* with one another.

A *transaction* is modeled to be an atomic sequence of methods issued to objects [1]. Multiple transactions are concurrently performed in order to increase the throughput of the system. Multiple conflicting transactions are required to be *serializable* to keep objects mutually consistent [1, 3]. Let T_i be a transaction which issues a method op_{1i} to an object o_1 and a method op_{2i} to another object o_2. Suppose there are a pair of transactions T_1 and T_2 where op_{11} and op_{21} conflict on the object o_1 as well as the methods op_{12} and op_{22} on the object o_2. If the method op_{11} is performed on the object o_1 before op_{21}, op_{21} is required to be performed before op_{22} on the other object o_2 according to the serializebility theory [1]. In the timestamp ordering (TO) scheduler [1], each transaction T_i is assigned with real time $ts(T_i)$ when the transaction T_i is initiated on a client. If $ts(T_1) < ts(T_2)$, the method op_{11} is performed before op_{21} on the object o and the method op_{12} is performed before op_{22} on the object o_2. Thus, a pair of conflicting transactions T_1 and T_2 are performed in the timestamp order.

In the two-phase locking protocol [3], the transaction T_1 is performed if a pair of the objects o_1 and o_2 are locked before the other transaction T_2. The transaction T_2 cannot manipulate the objects o_1 and o_2 until the transaction T_1 releases the objects. In the strict protocol [1], every transaction releases all the objects locked on termination of the transaction. Hence, no cascading abort occur.

2.2 Roles

In access control models [6], a system is composed of two types of entities, *subject* and *object*. A subject is an active entity which issues a request to an object like user and program. On the other hand, an object is a passive entity like database which receives a request and then sends back its response. A subject can manipulate an object only through a method which the subject is allowed to issue. An *access right* is a pair $\langle o, op \rangle$ of an object o and a method op. Only if an access right $\langle o, op \rangle$ is granted to a subject s, the subject s is allowed to manipulate an object o through a method op.

A *role* shows a job function in an enterprise. Each subject s plays a role like *president* in an enterprise. A subject which plays a more significant role should be more prioritized than less significant subjects. If a pair of tasks in different jobs would like to 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 a role-based access control (RBAC) model [6]. A subject s is first granted a role R. Then, the subject is allowed to issue an access request op to an object o only if an access right $\langle o, op \rangle$ is included in the role R. Suppose a subject s initiates a transaction T with a role R granted to the subject s. We assume each transaction is associated with only one role in this paper. Here, let $subject(T)$ denote a subject which initiates a transaction T. Let $role(T)$ show a role which is associated to a transaction T. A transaction T issues an access request $\langle o, op \rangle$ to manipulate an object o through a method op. The request $\langle o, op \rangle$ is accepted if $\langle o, op \rangle \in role(T)$. Otherwise, the access request $\langle o, op \rangle$ is rejected, i.e. the transaction T is aborted.

The relational database systems take the *discretionary* approach [5, 7]. A role R is first created by a subject s_0. Here, the subject s_0 is an *owner* of the role R, denoted by $owner(R)$. Then, the owner s_0 grants the role R to a subject s_1. Furthermore, the subject s_1 can grant the role R to another subject s_2. A role is also an object with methods *grant* and *revoke* for granting and revoking and methods *delete* and *add* for deleting and adding access rights in the role, respectively. If the subject s_1 changes the role R, e.g. adds an access right to R, the role R granted to the subjects s_0 and s_2 is also changed.

3 Significancy on Roles

3.1 Significancy of subjects on a role

We take the discretionary approach to adopting the role-based access control (RBAC) model [6] to object-based systems. First, suppose that a subject s_0 creates a role R. Here, the subject s_0 is an owner $owner(R)$ of the role R. Then, the owner subject s_0 grants the role R to another subject s_1. The subject s_1 furthermore grants the role R to subjects s_2 and s_3 as shown in Figure 1. Here, the subject s_1 is more tightly related with the role R than the subject s_2. This means, the subject s_1 is considered to be more significant than the other subject s_2 with respect to the role R.

We define a precedent relation among subjects showing which subjects are more significant than others with respect to a role R :

- A subject s_1 is more *significant* than another subject s_2 with respect to a role R ($s_1 \succ_R s_2$) if and only if (iff) the subject s_1 grants the role R to s_2 or $s_1 \succ_R s_3 \succ_R s_2$ for some subject s_3.

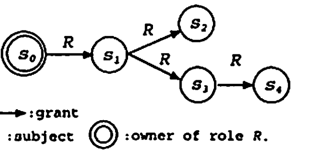

Figure 1. Discretionary approach.

The significantly precedent relation $\succ_R$ of subjects is acyclic. A pair of subjects s_1 and s_2 are *independent* with respect to a role R ($s_1 \parallel_R s_2$) iff s_1 and s_2 are granted the role R and neither $s_1 \succ_R s_2$ nor $s_2 \succ_R s_1$. In Figure 1, an owner subject s_0 ($owner(R)$) of a role R is more significant than a subject s_1 ($s_0 \succ_R s_1$) since the owner s_0 grants the role R to the subject s_1. In addition, $s_1 \succ_R s_2$ and $s_1 \succ_R s_3$. Thus, $s_0 \succ_R s_1 \succ_R s_2$ and $s_0 \succ_R s_2$. However, $s_2 \parallel_R s_3$ and $s_2 \parallel_R s_4$.

Let $\mathsf{S}(R)$ be a set of subjects which are granted a role R. Subjects in the set $\mathsf{S}(R)$ are partially ordered in the significantly precedent relation $\succ_R$. Suppose the role R includes a pair of access rights $\langle o, op_1 \rangle$ and $\langle o, op_2 \rangle$ where a method op_1 conflicts with a method op_2. A pair of the subjects s_1 and s_2 are granted the role R and issue methods op_1 and op_2 to the object o, respectively. If the subject s_1 is more significant than the subject s_2 with respect to the role R ($s_1 \succ_R s_2$), the method op_1 is performed before another method op_2 on the object o.

3.2 Significancy of roles

We discuss which roles are more significant than other roles. Suppose a subject s_1 is granted a role R_1 and a subject s_2 is granted another role R_2. Then, a pair of the subjects s_1 and s_2 issue conflicting methods op_1 and op_2 to an object o, respectively. We discuss which method op_1 or op_2 to be performed on the object o before the other method. It is true that op_1 should be performed before op_2 if a job function shown by a role R_1 is more significant than another role R_2 in an enterprise.

A method op_1 is more *significant* than another method op_2 on an object o ($op_1 \succ op_2$) iff the state of the object o is changed by the method op_1 but is not changed by the method op_2. Methods by which state of an object is changed are referred to as *object* methods. Object methods are classified into two types : *output* and *input* ones. By using an *output* type of method, data is derived from an object while an *input* type of method brings data into an object. Furthermore, there are *class* methods where an object is created for a class and is dropped. A pair of methods *create* and *drop* of an object are more significant than the object methods.

Let us consider a pair of methods *withdraw* and *deposit* on a *bank* object. Both the methods *withdraw* and *deposit* are input types. Hence, the methods *withdraw* and *deposit* are significantly equivalent (*withdraw* $\equiv$ *deposit*). In our life, a subject more carefully issues a method *withdraw* than a method *deposit* because the account value in the *bank* object is decremented by *withdraw*. This example shows that some methods are considered to be more significant than other methods by an application. Here, a method *withdraw* is referred to as *more semantically significant* than another method *deposit* (*withdraw* $\succ\!\!\succ$ *deposit*). A semantically significant relation $\succ\!\!\succ$ among methods is defined on each object by an application. A method op_1 is referred to as *semantically significantly equivalent* with another method op_2 ($op_1 \cong op_2$) iff neither $op_1 \succ\!\!\succ op_2$ nor $op_2 \succ\!\!\succ op_1$. $op_1 \succeq\!\!\succeq op_2$ iff $op_1 \succ\!\!\succ op_2$ or $op_1 \cong op_2$.

[Definition] A method op_1 is *more significant* than another method op_2 ($op_1 \succ op_2$) 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 is an *input* type and op_2 is an *output* one.
3. op_1 and op_2 are same types and op_1 is semantically more significant than op_2 ($op_1 \succ\!\!\succ op_2$).

A method op_1 is *significantly equivalent* with another method op_2 ($op_1 \equiv op_2$) iff neither $op_1 \succ op_2$ nor $op_2 \succ op_1$. A method op_1 *significantly dominates* another method op_2 ($op_1 \succeq op_2$) iff $op_1 \succ op_2$ or $op_1 \equiv op_2$.

A system is composed of multiple objects. 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 neither $o_1 \succ o_2$ nor $o_2 \prec o_1$. $o_1 \equiv o_2$ if $o_1 = o_2$. An object o_1 *significantly dominates* another object o_2 ($o_1 \succeq o_2$) iff $o_1 \succ o_2$ or $o_1 \equiv o_2$.

A role is a collection of access rights. Let $\langle o_1, op_1\rangle$ and $\langle o_2, op_2\rangle$ be access rights on a pair of objects o_1 and o_2. We discuss which one in the access rights $\langle o_1, op_1\rangle$ and $\langle o_2, op_2\rangle$ is more significant than the other. First, methods op_1 and op_2 are supported by a same object o_1 ($o_1 = o_2$). An access right $\langle o_1, op_1\rangle$ is more significant than $\langle o_1, op_2\rangle$ ($\langle o_1, op_1\rangle \succ \langle o_1, op_2\rangle$) if $op_1 \succ op_2$. Next, a pair of methods op_1 and op_2 are supported by different objects o_1 and o_2, respectively ($o_1 \neq o_2$). An access right $\langle o_1, op_1\rangle$ is more significant than another access right $\langle o_2, op_2\rangle$ ($\langle o_1, op_1\rangle \succ \langle o_2, op_2\rangle$) if $o_1 \equiv o_2$ and $op_1 \succ op_2$. Lastly, suppose that an object o_1 is more significant than another object o_2 ($o_1 \succ o_2$). An access right $\langle o_1, op_1\rangle$ is more significant than another access right $\langle o_2, op_2\rangle$ ($\langle o_1, op_1\rangle \succ \langle o_2, op_2\rangle$) if $o_1 \succ o_2$.

[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 one of the following condition holds:

- $op_1 \succ op_2$ if $o_1 \equiv o_2$.
- $o_1 \succ o_2$.

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 neither $\langle o_1, op_1\rangle \succ \langle o_2, op_2\rangle$ nor $\langle o_1, op_1\rangle \prec \langle o_2, op_2\rangle$. An access right $\langle o_1, op_1\rangle$ *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 $\langle o_1, op_1\rangle \equiv \langle o_2, op_2\rangle$.

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 $\langle o_3, op_3\rangle$ 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 role R_1 is more significant than another role R_2 ($R_1 \succ R_2$) iff $R_1 \succeq R_2$ but $R_1 \not\equiv R_2$. A pair of roles R_1 and R_2 are *comparable* if $R_1 \succeq R_2$ or $R_2 \succeq R_1$. Otherwise, R_1 and R_2 are uncomparable.

4 Serializability

Suppose a pair of transactions T_1 and T_2 are granted roles R_1 and R_2, respectively. Each transaction is submitted by a subject and assigned with one of roles granted to the subject. Let $\mathbf{T}$ be a set of transactions which are being performed in a system. The transaction set $\mathbf{T}$ is partially ordered based on the significantly dominant relation $\succeq$ of roles:

[Definition] A transaction T_1 *significantly dominates* another transaction T_2 ($T_1 \succeq T_2$) iff $role(T_1) \succeq role(T_2)$ or $subject(T_1) \succeq_R subject(T_2)$ if $role(T_1) = role(T_2) = R$.

A transaction T_1 is *significantly equivalent* with another transaction T_2 ($T_1 \equiv T_2$) if $T_1 \succeq T_2$ and $T_2 \succeq T_1$. T_1 and T_2 are *independent* iff neither $T_1 \succeq T_2$ nor $T_2 \succeq T_1$.

A *schedule* H is an execution sequence of methods from transactions in the transaction set $\mathbf{T}$. A transaction T_1 *precedes* another transaction T_2 in the schedule 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 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 $T_1 \succeq T_2$.

Suppose a transaction T_1 precedes another transaction T_2 in a schedule H of a transaction set $\mathbf{T}$. Here, if $T_1 \succeq T_2$, "$T_1 \rightarrow_H T_2$" is referred to as *legal*, i.e. T_1 significantly precedes T_2 ($T_1 \Rightarrow_H T_2$). That is, conflicting transactions are performed in the significantly precedent relation $\Rightarrow_H$. On the other hand, if $T_1 \prec T_2$, "$T_1 \rightarrow_H T_2$" is *illegal*. A schedule H, i.e. $\langle \mathbf{T}, \rightarrow_H\rangle$ is *legal* iff $T_1 \rightarrow_H T_2$ if $T_1 \succeq T_2$ for every pair of transactions T_1 and T_2 in $\mathbf{T}$. In order to make a schedule *legal*, methods from transactions are required to be buffered until all the transactions are initiated. Here, the throughput of the system is degraded since transactions have to wait in the buffer. In order to increase the throughput, only some number of transactions in $\mathbf{T}$ which are initiated during some time units are scheduled. A schedule H is partitioned into subschedules $H_1, ..., H_n$ where each subschedule $H_i = \langle \mathbf{T}_i, \rightarrow_H \rangle$ ($i = 1, ..., n$) satisfies the following conditions:

[Role ordering (RO) partition]

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. $T_1 \rightarrow_H T_2$ is legal if $T_1 \rightarrow_H T_2$ for every pair of transactions T_1 and T_2 in $\mathbf{T}_i$.
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 $T_{j2} \rightarrow_H T_{i2}$.

Figure 2 shows a hasse diagram of a schedule H for a transaction set $\mathbf{T} = \{T_1, T_2, T_3, T_4, T_5, T_6,\}$ where a directed edge from a transaction T_i to T_j shows $T_i \rightarrow_H T_j$. $\Rightarrow$ and $\rightarrow$ show legal and illegal precedent relation $\rightarrow_H$. Suppose that $T_1 \succeq T_2$, $T_3 \succeq T_2$, $T_4 \succeq T_5$, $T_4 \succeq T_6$, $T_4 \succeq T_2$, and $T_6 \succeq T_3$. Here, a pair of subschedules H_1 with $\mathbf{T}_1 = \{T_1, T_2, T_3\}$ and H_2 with $\mathbf{T}_2 = \{T_4, T_5, T_6\}$ are RO partitions of the schedule H. In the schedule H_1, methods from the transactions T_1, T_2, and T_3 are first performed in the significantly dominant relation $\succeq$, i.e. $T_1 \Rightarrow_H T_2$ and $T_3 \Rightarrow_H T_2$. Since $T_2 \preceq T_4$ and $T_3 \preceq T_6$, the transactions T_4 and T_6 cannot be performed as long as every transaction in H_i completes. After T_2 commits, the transactions in H_2 are performed.

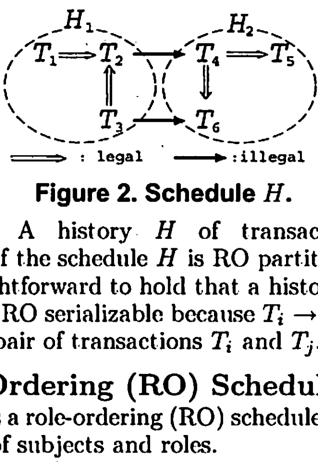

Figure 2. Schedule H.

[Definition] A history H of transactions is *RO serializable* if the schedule H is RO partitioned.

It is straightforward to hold that a history H is serializable if H is RO serializable because $T_i \rightarrow_H T_j$ if $T_i \Rightarrow_H T_j$ for every pair of transactions T_i and T_j.

5 Role-Ordering (RO) Scheduler

We discuss a role-ordering (RO) scheduler based on the significancy of subjects and roles.

5.1 One-object model

First, we discuss a role-ordering (RO) scheduler for a single object which is manipulated by multiple transactions. An object is stored in an object base (*OB*) of a server. Multiple transactions on clients issue methods to an object o. A transaction issues a *commit* (c) or *abort* (a) method at the end. An RO scheduler is composed of a receipt queue *RQ* and auxiliary receipt queue *ARQ*. On receipt of a method from a transaction, the method is first enqueued in *RQ* of the object o [Figure 3]. 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) : a method op is enqueued into Q.
2. op := **dequeue**(Q) : a method op is dequeued from Q.
3. op := **top**(Q) : a method op is a top method in Q.
4. **ROsort**(Q) : all methods in Q are sorted in the significantly dominant relation $\succeq$ of transactions.

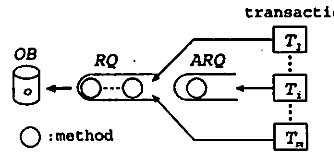

Figure 3. RO scheduler.

A variable $\mathbf{E}$ shows a set of methods being currently performed on an object o. Let $\mathbf{TE}$ be a set of transactions being currently performed, i.e. $\{Tr(op) \mid op \in \mathbf{E}\}$. A variable $\mathbf{C}$ denotes a transaction which is performed on the object o and which is significantly dominated by every transaction performed. Initially, $\mathbf{C} := \top$. Here, $\top$ and $\bot$ denote *top* and *bottom* transactions, respectively, where $\top \succeq T \succeq \bot$ for every transaction T. 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 in $T_1, ..., T_m$ being performed are stored in $\mathbf{E}$. Here, $\mathbf{C}$ shows a transaction T_i where $T_i \preceq T_j$ for every $j = 1, ..., m$. If $T \succeq \mathbf{C}$, the method op is enqueued into *RQ*. However, if $T \succ \mathbf{C}$, the method op is enqueued into *ARQ*. After that, every method issued to the object o from every transaction not in $\mathbf{E}$ is enqueued into *ARQ*. If every transaction in $\mathbf{E}$ commits or aborts, i.e. $\mathbf{E}$ is empty, all methods in *ARQ* are moved to *RQ*. That is, one subschedule is finished and a new schedule is started. Then, methods in *RQ* are sorted in the significantly dominant relation $\succeq$.

[Delivery of a method op from a transaction T]

> **if** $T \in \mathbf{TE}$ **or** $T \preceq \mathbf{C}$, { **enqueue**(op, RQ); **ROsort**(RQ); }
> **else** { $\mathbf{C} := \bot$; **enqueue**(op, ARQ); }

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

[Execution of methods]

1. **if** $\mathbf{TE} = \phi$, { $\mathbf{C} := \top$; Every method op in *ARQ* is moved to *RQ*; **ROsort**(*RQ*); **goto** 1; }
2. **if conflict**(op, $\mathbf{E}$), **return**;
   else{ op : = **dequeue**(*RQ*);
   if $Tr(op) \notin \mathbf{TE}$, $\mathbf{TE} := \mathbf{TE}$ *cup* $\{Tr(op)\}$;
   $\mathbf{E}$: = $\mathbf{E} \cup \{op\}$; **if** $Tr(op) \prec \mathbf{C}$, $\mathbf{C} := Tr(op)$;
   perform(op);}

Let op be a method on an object o, which is the top in *RQ*. If the method op is compatible with every method being currently performed, the top method op is dequeued from *RQ* and then is performed on the object o in *OB*. Otherwise, no method in *RQ* is dequeued.

If a method 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 conflicts is kept waited in RQ, every other method in RQ is required to be waited. Here, suppose there is another method op_2 following the method op_1 in RQ. If op_2 is compatible with op_1, op_2 can be performed by jumping ever op_1 in RQ.

[Definition] A method op is referred to as *ready* in a receipt queue RQ iff op is compatible with every method preceding op in RQ and with every method in **E**.

In the execution procedure, if the top method op (= **top**(RQ)) cannot be performed, ready methods in RQ are taken in the significantly dominant relation $\succeq$ and then performed. We introduce the following procedures :

- **ready**(op, RQ, **E**) : ***true*** if a method op is ready in the receipt queue RQ, else ***false***.
- op_1: = **next**(op, RQ) : op_1 is a method in the receipt queue 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, i.e. **conflict**(op, **E**) is true, the following procedure is performed:

op : = **top**(RQ);
if conflict(op, **E**), {
op : = **next**(op, RQ);
while($op \neq$ NULL) {
  if ready(op, RQ, **E**), {
    op is removed from RQ; **E** : = **E** $\cup$ $\{op\}$;
    TE : = **TE** $\cup$ $\{Tr(op)\}$ **if** $Tr(op) \notin$ **TE**;
    if $Tr(op) \prec$ **C**, **C** := $Tr(op)$;
    perform(op); **break**; }
  else op : = **next**(op, RQ); } }

[Theorem] A schedule of a transaction set **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 **T** is RO partitioned into the subsequences.

5.2 Distributed object model

In a distributed model, there are multiple objects o_1, ..., o_m ($m > 1$) distributed in servers and multiple transactions T_1, ..., T_l ($l > 1$) on multiple clients c_1 ..., c_n ($n > 1$). Each object o_i receives methods from multiple transactions on clients c_1 ..., c_n while each transaction issues methods to multiple objects.

There are local receipt queues RQ_{i1}, ..., RQ_{in} in each object o_i ($i = 1, \dots, m$). Transactions are initiated on a client c_s and issue methods to objects in servers. Methods issued from transactions 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, i.e. an object o_i receives every message from each client c_s in the sending order and with neither message loss nor duplication.

Requests in local receipt queues RQ_{i1}, ..., RQ_{in} are moved to a global receipt queue RQ_i in an object o_i [Figure 4]. Here, requests in the global receipt queue RQ_i are sorted in the significantly dominant relation $\succeq$. Then, the top method in the global receipt queue RQ_i is dequeued and then is performed if no method conflicting with the top method is currently being performed. Question is when the top method in the global receipt queue RQ_i can be dequeued. Let us consider a pair of transactions T_1 and T_2 as shown in Figure 4. The transaction T_1 issues a pair of methods op_{11} and op_{12} to the objects o_1 and o_2, respectively. The transaction T_2 issues a pair of methods op_{21} and op_{22} to the objects o_1 and o_2, respectively. Suppose a pair of the methods op_{11} and op_{21} conflict on the object o_1 and a pair of the methods op_{12} and op_{22} also conflict in the other object o_2. Suppose a method op_{12} is delayed and another method op_{22} is also delayed due to congestions and faults. In the object o_1, the method op_{11} is enqueued into the global receipt queue RQ_1 from the local receipt queue RQ_{11}, and then performed. On the other hand, the method op_{22} is performed in the object o_2 as well. Eventually, a pair of the delayed methods op_{21} and op_{12} arrive at the objects o_1 and o_2, respectively, and then are performed. Here, a pair of the transactions T_1 and T_2 are not serializable.

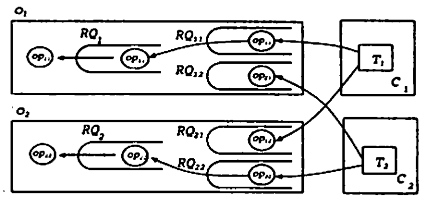

Figure 4. Schedulers.

The following conditions have to be satisfied for a collection of global receipt queues RQ_1, ..., RQ_m for objects o_1, ..., o_m, respectively, to realize the serializability of multiple transactions :

[Role-based serializability (RBS) conditions]

1. Methods in every global receipt queue RQ_i are sorted in the significantly dominant relation $\succeq$ of transactions ($i = 1, ..., m$).
2. For a top method op_s from a transaction T_s in each global receipt queue RQ_i, if there is a method $\acute{op}_t$ from the transaction T_t in RQ_i which the method op_s precedes and conflicts with op_t, $\acute{op}_s$ precedes op_t in every global receipt queue RQ_j where op_t and op_s are methods form T_t and T_s, respectively, and $\acute{op}_s$ and op_t conflict with one another

The second RBS condition shows the traditional serializability in a distributed database system [3]. The first condition means that every pair of conflicting methods are performed in the significantly dominant relation $\succeq$ of the transactions.

In order to satisfy the RBS conditions, we take the following approach :

1. Each client c_s periodically sends a *fence* message k_s to every object o_i.
2. In an object o_i, if there is a fence message k_s in every local receipt queue RQ_{is}, methods preceding a *fence* massage k_s in every local receipt queue RQ_{is} are moved to the global receipt queue RQ_{is}. Then, a *fence* message k_s is dequeue from the local receipt queue RQ_{is}. Finally, a *fence* message k_s is enqueued into the global receipt queue RQ_i.
3. Methods from the *fence* method or the top method to the *fence* message just enqueued are sorted in the significantly dominant relation $\succeq$.
4. A top method in the global receipt queue RQ_i is performed according to the execution procedure.

6 Evaluation

We implemented the RO scheduler and the locking system with deadlock detection. We evaluate the role ordering (RO) scheduler for a single object in terms of computation time of each method compared with the traditional two-phase locking (2PL) protocol. Transactions in clients issue methods to the RO scheduler and the locking module on an object base.

In the evaluation, an object o supports ten types of methods. We assume it takes same time to perform every method. We assume one method can be performed for one time unit if there is no other transaction. If multiple conflicting methods are concurrently performed, a method op has to wait until methods conflicting with op complete. If deadlock is detected in the locking protocol, methods performed in a transaction are undone if the transaction is aborted to release the deadlock. This means, it takes longer to perform a method than one time unit. The computation ratio τ is defined to be the ratio of the total number of methods effectively performed to the total processing time units. If all the transactions are serially performed, the computation ratio τ is 1.0 which is the maximum. $\tau = 0$ if no method is performed, e.g. every transaction is deadlocked and aborted. A conflicting relation on the methods is randomly defined so that each method averagely conflicts with 10 % of the other methods. There are five roles R_1, ..., R_5. Each role R_i includes three access rights, which are randomly selected out of ten possible access rights on the object o.

There are three subjects s_0, s_1, and s_2. The subject s_0 is an owner of the roles R_1, ..., R_5. The subject s_0 grants each role to the other subjects. That is, $s_0 \succeq_{R_i} s_1$, $s_0 \succeq_{R_i} s_2$, and $s_0 \|_{R_i} s_2$ for every role R_i (i = 1, ..., 5). The roles are ordered as $R_1 \succeq R_2 \succeq R_3$, $R_1 \succeq R_4 \succeq R_5$, $R_2 \equiv R_4$, $R_2 \equiv R_5$, $R_3 \equiv R_4$, and $R_3 \equiv R_5$.

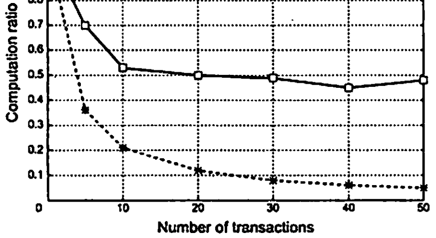

Figure 5. Evaluation of one-object model.

A transaction issues five methods randomly selected from the ten methods of the object, where some method may be invoked multiple times. A role is also randomly assigned to each transaction.

For each configuration, i.e. object, roles and transactions generated based on the random number, the computation ratio τ is calculated multiple times in the simulation until the average value of the computation ratio is saturated. Figure 5 shows the computation ratio τ for the number of transactions. The computation ratio τ = 1.0 shows the maximum ratio. As shown in Figure 5, the RO scheduler implies higher throughput than the 2PL protocol. For example, the RO scheduler implies six times and ten times higher throughput than the 2PL protocol for 20 and 40 transactions, respectively.

Figures 6 and 7 show average values of processing time of the RO scheduler and the 2PL protocol, respectively, for the total number of transactions. The processing time shows time units from time when a method in each transaction which is assigned with a role R_i (i = 1, ..., 5) issued to time when the method completes. In the RO scheduler, a transaction T_i which is assigned with a more significant role than another transaction T_j can manipulate an object o earlier than transactions with less significant roles. On the other hand, the computation order of transactions is independent of the significancy of roles in the 2PL protocol.

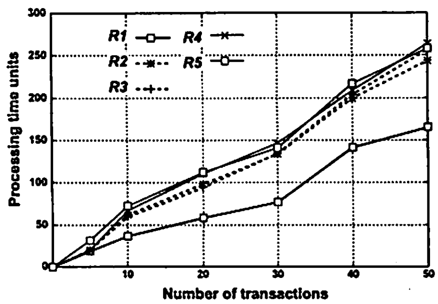

Figure 6. RO scheduler.

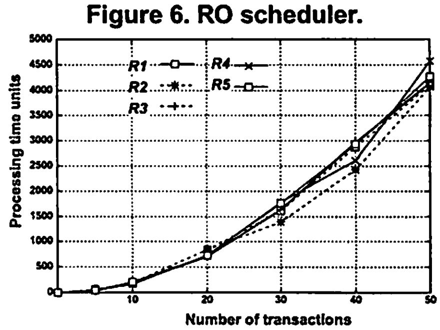

Figure 7. Two-phase locking (2PL) protocol.

7 Concluding Remarks

We discussed a role ordering (RO) scheduler based on role concept in this paper. 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 role ordering (RO) scheduler for single-server and multi-server models and how to implement the *RO* scheduler. Conflicting methods from multiple transactions are performed in the significantly dominant relation. That is, the more significant role a transaction is assigned, the earlier methods from the transaction are performed.

We evaluated the RO scheduler compared with the traditional two-phase locking protocol (2PL). In the evaluation, we showed the RO scheduler can support higher throughput than the 2PL protocol.

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] J. Gray. Notes on Database Operating Systems. *Lecture Notes in Computer Science*, (60):393–481, 1978.

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

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

[6] 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.

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

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