Vol. 46 No. 12 IPSJ Journal Dec. 2005

Recommended Paper

Role Ordering (RO) Scheduler for Distributed Objects

Tomoya Enokido†

A role-based access control model is used to make a system secure. A role concept shows a job function in an enterprise. 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, we discuss a concurrency control based on the significancy of roles assigned to transactions. We define a significantly dominant relation on roles. We discuss a role ordering (RO) scheduler based on the role concept. We evaluate the RO scheduler compared with the two-phase locking (2PL) protocol.

1. Introduction

Information systems like database systems [10),13)] adopt role-based access control (RBAC) models [3),6),9),12),15)] to make the systems secure. A *role* shows a job function like president and secretary, which each person performs in an enterprise. We use word "role" in software engineering field. A role is a set of access rights. An *access right* is realized in a pair $\langle o, op\rangle$ of an object o and a method op. Only if a role R which includes an access right $\langle o, op\rangle$ is granted to a subject, the subject is allowed to manipulate the object o through the method op. In the *discretionary* approach [10),13)], 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),7)]. 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* 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 mutually consistent. In order to realize the serializability of multiple conflicting transactions, locking protocols [1),7)] are widely used. 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)]. By using the TO scheduler, transactions are totally ordered in their timestamps.

In this paper, we discuss a *role ordering* (*RO*) scheduler to make a set of transactions serializable based on roles associated for transactions. Each job is realized to be a collection of transactions in an enterprise. 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, i.e. a job function shown by R_1 is more significant than R_2 in an enterprise. 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 define significantly dominant relations among roles. In Section 4, we discuss the role ordering (RO) serializability. In Section 5, we discuss the 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

An object-based system is composed of objects [8)] 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

† Faculty of Business Administration, Rissho University

The initial version of this paper was presented at the DPS workshop (DPSWS12) held on Dec. 2004, which was sponsored by SIGDPS. This paper was recommended to be submitted to IPSJ Journal by the program chair of DPSWS12.

the methods depends on the computation order. Otherwise, a pair of the methods op_1 and op_2 are *compatible* ($op_1 \Box op_2$).

A *transaction* is an atomic sequence of methods[1]. Multiple transactions are concurrently performed on objects. Multiple conflicting transactions are required to be *serializable* to keep objects mutually consistent[1,7]. Let T_i be a transaction which issues a method op_{1i} to an object o_1. Suppose there are a pair of transactions T_1 and T_2 where op_{11} and op_{12} conflict on the object o_1 as well as the methods op_{21} 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]. Let $\mathbf{T}$ be a set of transactions $\{T_1, ..., T_n\}$. Let H be a schedule of transactions in $\mathbf{T}$, i.e., sequence of methods performed. A transaction T_i *precedes* another transaction T_j ($T_i \rightarrow_H T_j$) in H iff a method op_i from T_i is performed before a method op_j from T_j where $op_i \diamond op_j$. A schedule H is serializable iff the precedent relation $\rightarrow_H$ is acyclic. In the timestamp ordering (TO) scheduler[1], each transaction T_i is assigned with time $ts(T_i)$ showing what time the transaction T_i is initiated on a client. A pair of conflicting methods issued by transactions T_1 and T_2 are performed in the timestamp order. In the two-phase locking (2PL) protocol[7], 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.

2.2 Roles

In access control models[1,2,4,6,11,12,14,15], a system is composed of two types of entities, *subject* and *object*. A subject manipulates an object. A *role* shows a job function in an enterprise like *president*. Each subject s plays a role. A subject which plays a more significant role should be more prioritized than less significant subjects. A task is realized as a transaction. 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 role is a collection of *access rights* in a role-based access control (RBAC) model[12]. An access right is a pair $\langle o, op \rangle$ of an object o and a method op. 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 R. We assume each transaction is associated with only one role in this paper. Let $subject(T)$ denote a subject which initiates a transaction T. Let $role(T)$ show a role which is associated to a transaction T.

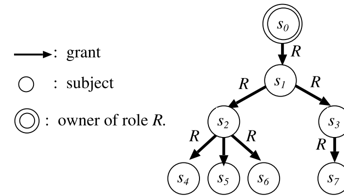

Fig. 1 Discretionary approach.

3. Significancy on Roles

3.1 Significancy of Subjects on a Role

The relational database systems take the *discretionary* approach[10,13]. We take the discretionary approach to adopting the role-based access control (RBAC) model[12] to object-based systems. First, suppose that a subject s_0 creates a role R. Here, the subject s_0 is an *owner* of the role R, denoted by $owner(R)$. Then, the subject s_0 grants the role R to another subject s_1. The subject s_1 furthermore grants the role R to a pair of subjects s_2 and s_3 (**Fig. 1**). If the subject s_1 changes the role R, the role R granted to the subjects s_0 and s_2 is also changed.

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 the subject s_2 or $s_1 \succ_R s_3 \succ_R s_2$ for some subject s_3.

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 $s_2 \succ_R s_1$. Because, there are no *significant* relations with respect to a role R ($\succ_R$) which is defined in above between subjects s_1 and s_2.

3.2 Significancy of Roles

Next, we discuss which roles are more significant than other roles. 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 de-

riving data from an object. On the other hand, a *change* type of method is one for changing a state of an object.

Let us consider a pair of change methods *withdraw* and *deposit* on a *bank* object. 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 through *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 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 $op_2 \succ\!\!\succ op_1$. A method op_1 semantically significantly dominates a method op_2 ($op_1 \succeq\!\!\succeq op_2$) 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 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 $op_1 \succ\!\!\succ op_2$.

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 $op_1 \equiv op_2$.

Objects are classified into some security classes [4),5)]. 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. 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 $o_2 \prec o_1$. An object o_1 *significantly dominates* another object o_2 ($o_1 \succeq o_2$) iff $o_1 \succ o_2$ or $o_1 \equiv o_2$.

Next, 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.

[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 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 $\langle o_1, op_1 \rangle \equiv \langle o_2, op_2 \rangle$.

Finally, 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 least upper bound (*lub*) $R_1 \bigcup R_2$ 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*) $R_1 \bigcap R_2$ is similarly defined.

4. Serializability

Let $\mathbf{T}$ be a set of transactions which are being performed in a system. We define which transaction T_1 or T_2 in $\mathbf{T}$ is significant based on the significantly dominant relations of subjects and 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 $subject(T_1) \succ_R subject(T_2)$ 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 $T_2 \succeq T_1$. A least upper bound (*lub*) $T_1 \bigcup T_2$ of transactions T_1 and T_2 is a transaction T_3 where $T_3 \succeq T_1$ and $T_3 \succeq T_2$ and there is no transaction T_4 such that $T_3 \succeq T_4 \succeq T_1$ and $T_3 \succeq T_4 \succeq T_2$. A greatest lower bound (*glb*) $T_1 \bigcap T_2$ is defined similarly. We assume that a top transaction $\top$ and a bottom transaction $\bot$ exist where $\top \succeq T \succeq \bot$ 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 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 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 $T_1 \succeq T_2$, i.e. $T_1 \Rightarrow_H T_2$ if op_1 and op_2 conflict and op_1 is performed before op_2 for every pair of methods op_1 and op_2 from T_1 and T_2, respectively.

Suppose a transaction T_1 precedes another transaction T_2 in a schedule H. Here, if $T_1 \succeq T_2$, a precedent relation "$T_1 \rightarrow_H T_2$" is *legal* in a schedule H. On the other hand, if $T_1 \prec T_2$, "$T_1 \rightarrow_H T_2$" is *illegal* in a schedule H. A schedule $H = \langle \mathbf{T}, \rightarrow_H \rangle$ 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 a schedule 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 $T_1 \succeq T_2$, op_1 is performed before op_2.

In order to make a schedule *legal*, methods from transactions are required to be buffered until all the transactions are initiated. However, the throughput of the system is degraded. 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 $H = \langle \mathbf{T}, \rightarrow_H \rangle$ 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 $\mathbf{T}_1 \cup \cdots \cup \mathbf{T}_n = \mathbf{T}$. That is, every pair of T_i and T_j in $\mathbf{T}$ are independent.
2. A precedent relation $T_1 \rightarrow_H T_2$ is legal in a schedule H if $T_1 \rightarrow_{H_i} T_2$ for every pair of transactions T_1 and T_2 in $\mathbf{T}_i$ of a subschedule 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 $T_{j2} \rightarrow_H T_{i2}$.

A role is assigned to each transaction when the subject initiates the transaction. Significantly dominant relations among transactions are defined based on the significancy of roles assigned to each transaction, which we defined at the beginning of this section. Here, suppose that six transactions are initiated (see **Fig. 2**) and a transaction T_1 significantly dominates a transaction T_2 ($T_1 \succeq T_2$), $T_3 \succeq T_2$, $T_4 \succeq T_5$, $T_4 \succeq T_6$, $T_4 \succeq T_2$, and $T_6 \succeq T_3$. In addition, suppose that a schedule H is RO partitioned into a pair of subschedules H_1 with $\mathbf{T}_1 = \{T_1, T_2, T_3\}$ and H_2 with $\mathbf{T}_2 = \{T_4, T_5, T_6\}$. If the subschedule H_2 is started before another subschedule H_1 is not completed, a pair of transactions T_2 and T_4 are concurrently performed and T_2 may be completed before T_4 ($T_2 \rightarrow_H T_4$) although $T_2 \preceq T_4$. Similarly, T_3 may be completed before T_6 ($T_3 \rightarrow_H T_6$) although $T_3 \preceq T_6$. Therefore, the schedule H may become illegal. In this case, 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 completes in the subschedule H_1.

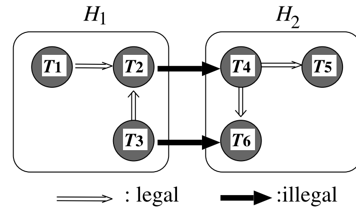

Fig. 2 Schedule H.

[Definition] Let $\mathbf{T}$ be a set of transactions. 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 a role-ordering (RO) scheduler for a single object. Multiple transactions on clients 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) : a method op is enqueued into a queue Q.
2. op := **dequeue**(Q) : a method op is dequeued from a queue Q.
3. op := **top**(Q) : a method op is a top method in a queue Q.
4. **ROsort**(Q) : all methods in a queue Q are sorted in the significantly dominant relation $\succeq$ 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 dominated by every transaction performed. Initially, $\mathbf{C} := \top$. There are following procedures to perform a method op on an object o:

1. **conflict**(op, **E**) : **false** if $\mathbf{E} = \phi$ or a method op does not conflict with every method in **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 **E**. Here, a variable **C** shows a transaction T_i where $T_i \preceq T_j$ for every i and j ($i = j = 1, ..., m$). If $T \preceq \mathbf{C}$, the method op is enqueued into the receipt queue RQ. However, if $T \succ \mathbf{C}$, op is enqueued into the auxiliary receipt queue ARQ.

[Delivery of a method op**]**

```
if T ∈ TE or T ⪯ C {
      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 Tr(op) ≺ C, C := Tr(op);
          perform(op); }
```

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

[Completion of method op**]**

1. **E** : = **E** - $\{op\}$;
2. **TE** := **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 a receipt queue RQ iff op is compatible with not only every method in being performed but also every waiting method preceding op in the receipt queue RQ.

We introduce the following procedures:

1. **ready**(op, RQ, **E**) : ***true*** if a method op is ready in the receipt queue RQ, else ***false***.
2. 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 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 Tr(op) ≺ 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 $o_1, ..., o_l$ ($l > 1$) distributed in multiple servers and multiple transactions $T_1, ..., T_m$ ($m > 1$) on multiple clients $c_1 ..., c_n$ ($n > 1$). A *request* is written in a pair $\langle o_i, op_i \rangle$ of an object o_i and a method op_i on the object o_i. 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 issues a *commit* (c) or an *abort* (a) method to every object in $oset(T_t)$.

Each client c_s has a variable f which is initialized by one ($f = 1$) ($s = 1, ..., n$). Each client

c_s periodically sends a $fence$ message which includes the variable f of c_s to all objects in a system. 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 transactions 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 the queue GRQ_i are sorted in the significantly dominant relation $\succeq$ of transactions. The following conditions have to be satisfied for a collection of global receipt queues GRQ_1, ..., GRQ_l for objects o_1, ..., o_l, respectively, to realize the serializability of multiple transactions:

[**Role-based serializability (RBS)**]

1. Methods in every GRQ_i are sorted in the significantly dominant relation $\succeq$ 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 the transaction T_u in GRQ_i which the method op_u precedes and conflicts with op_t, op'_u precedes op_t in every global receipt queue GRQ_j where op_t and op_u are methods form T_t and T_u, respectively, and op'_u and op_t conflict with one another.

We discuss how role ordering (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 variable f_{o_i} which was initialized to be one ($f_{o_i} = 1$).

[**Receiving procedure**]

If there is a $fence$ message k_s whose variable f is equal to a variable f_{o_i} of the object o_i ($f = f_{o_i}$) in every local receipt queue RQ_{is}, methods and $mset(T_t)$ preceding the $fence$ message k_s are dequeued from every local receipt queue RQ_{is}. Then, one of the following procedures is performed (**Fig. 3**).

1. If a dequeued message is $mset(T_t)$, $mset(T_t)$ is enqueued into an auxiliary global receipt queue ($AGRQ_i$) of o_i in significantly dominant relation of transaction T_t. Here, $mset(T_t)$ cannot be enqueued into $AGRQ_i$ by beyond a fence message which has already enqueued into $AGRQ_i$.
2. If a dequeued message is a method op_{it} issued by a transaction T_t to the object o_i, RO scheduler searches $AGRQ_i$ to confirm where the $mset(T_t)$ of the transaction T_t is enqueuing into. There are three cases. 1) If $mset(T)$ is enqueuing between the top of $AGRQ_i$ and the first fence message k_1 in $AGRQ_i$, the method op_{it} is stored between the top of GRQ_i and the first fence message k_1 of GRQ_i in the significantly dominant relation of transaction $\succeq$. 2) If $mset(T)$ is enqueuing between a pair of fence messages k_l and k_m of $AGRQ_i$, the method op_{it} is stored between a pair of fence messages k_l and k_m of GRQ_i in $\succeq$. 3) If $mset(T)$ is enqueuing between last fence message k_n in $AGRQ_i$ and the end of $AGRQ_i$, the method is enqueued between last fence message k_n of GRQ_i and the end of GRQ_i in $\succeq$.
3. If all top messages in every local receipt queue RQ_{i1}, ..., RQ_{in} are $fence$ message including the same value of f, a $fence$ message k is enqueued into GRQ_i and $AGRQ_i$ of the object o_i.

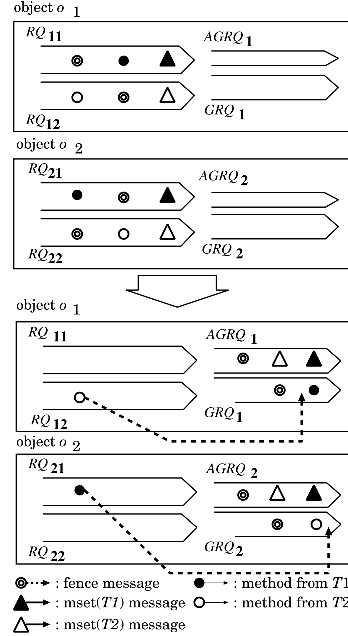

Fig. 3 State of local receipt queues.

Next, we discuss how role ordering (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, op_{it} is delivered to an object o_i.
   1.1 The method op_{it} does not conflict with every method which is 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 the currently performing methods on o_i. After all methods are completed, the $fence$ message k is removed from GRQ_i.

Next, we show a behavior of each object.

[**Completion of a method**]

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

6. Evaluation

We evaluate the role ordering (RO) scheduler for a single object system in terms of computation time of each method compared with the traditional two-phase locking (2PL) protocol. In the evaluation, an object o supports ten types of methods. We assume it takes a same time to perform every method. We assume one method can be performed for one time unit if there is no other transaction. 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. 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 (i = 1, ..., 5). 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 $s_1 \|_{R_i} s_2$ 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 $R_3 \equiv R_5$. A transaction issues five methods randomly selected from the ten methods of the object. A role is also randomly assigned to each transaction. The computation ratio τ is calculated multiple times in the simulation until the average value of the computation ratio is saturated.

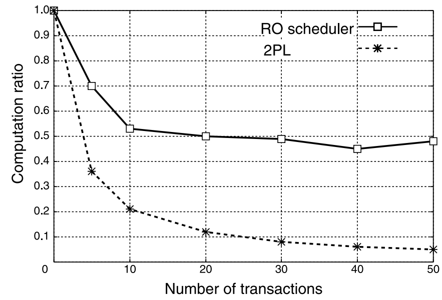

Fig. 4 Evaluation of one-object model.

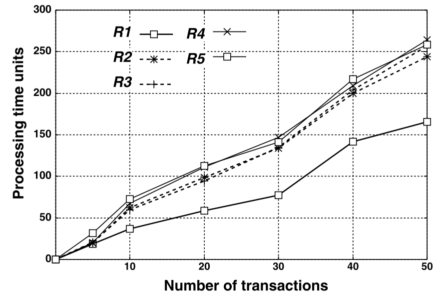

Fig. 5 RO scheduler.

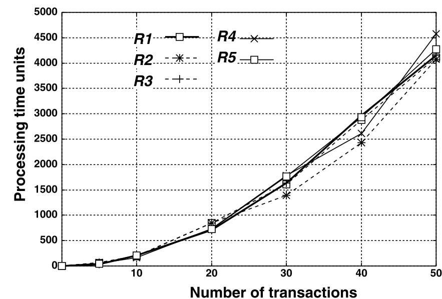

Fig. 6 Two-phase locking (2PL) protocol.

Figure 4 shows the computation ratio τ for the number of transactions. The RO scheduler implies higher throughput than the 2PL protocol. **Figures 5** and **6** 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 the duration [time unit] from time when a method in each transaction assigned with a role R_i (i = 1, ..., 5) is issued to time when the method completes. In the RO scheduler, a transaction T_i which is assigned with a more significant role 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.

7. Concluding Remarks

A *role* concept is widely used to design and implement information systems. The role concept shows a job function in an enterprise. In this paper, we discussed a concurrency control based on the significancy of roles assigned to transactions. We proposed a role ordering (RO) scheduler which serializes multiple conflicting transactions according to the significantly dominant relation of roles We discussed the RO scheduler for single-server and multi-server models and how to implement the RO scheduler. We showed the effectiveness of RO scheduler in terms of throughput and waiting time through the evaluation of RO scheduler compared with the traditional two-phase locking protocol (2PL).

References

1) Bernstein, P.A., Hadzilacos, V. and Goodman, N.: Concurrency Control and Recovery in Database Systems, *Addison-Wesley* (1987).
2) Bertino, E., Samarati, P. and Jaodia, S.: High Assurance Discretionary Access Control in Object Bases, *Proc. 1st ACM Conf. on Computers and Communication Security*, pp.140–150 (1993).
3) Chon, R., Enokido, T. and Takizawa, M.: Inter-Role Information Flow in Object-based Systems, *Proc. IEEE 18th International Conf. on Advanced Information Networking and Applications (AINA-2004)*, Vol.1, pp.196–201 (2004).
4) Denning, D.E.: A Lattice Model of Secure Information Flow, *Comm. ACM*, Vol.19, No.5, pp.236–343 (1976).
5) Denning, D.E. and Denning, P.J.: *Cryptography and Data Security*, Addison-Wesley Publishing Company (1982).
6) Ferraiolo, D. and Kuhn, R.: Role-Based Access Controls, *Proc. 15th NIST-NCSC National Computer Security Conf.*, pp.554–563 (1992).
7) Gray, J.: Notes on Database Operating Systems, *Lecture Notes in Computer Science*, No.60, pp.393–481 (1978).
8) Inc., O.M.G.: The Common Object Request Broker : Architecture and Specification, *Rev. 2.1* (1997).
9) Izaki, K., Tanaka, K. and Takizawa, M.: Information Flow Control in Role-Based Model for Distributed Objects, *Proc. IEEE International Conf. on Parallel and Distributed Systems (ICPADS-2001)*, pp.363–370 (2001).
10) Oracle Corporation: Oracle8i Concepts Vol. 1 (1999). Release 8.1.5.
11) Sandhu, R.S.: Lattice-Based Access Control Models, *IEEE Computer*, Vol.26, No.11, pp.9–19 (1993).
12) Sandhu, R.S., Coyne, E.J., Feinstein, H.L. and Youman, C.E.: Role-Based Access Control Models, *IEEE Computer*, Vol.29, No.2, pp.38–47 (1996).
13) Sybase: Sybase SQL Server. http://www.sybase.com/
14) Tachikawa, T., Yasuda, M. and Takizawa, M.: A Purpose-oriented Access Control Model in Object-based Systems, *Trans. IPSJ*, Vol.38, No.11, pp.2362–2369 (1997).
15) Tari, Z. and Chan, S.W.: A Role-Based Access Control for Intranet Security, *IEEE Internet Computing*, Vol.1, pp.24–34 (1997).

(Received April 22, 2005)
(Accepted September 2, 2005)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.1, pp.626–633.)

Editor's Recommendation

The author proposes a scheduling method based on role for distributed objects. In a sense, role may be similar to priority. They are useful to schedule objects in a suited order. The author also compares the performance of the proposed method with a conventional method and shows that the performance is clearly improved.
(Program chair of DPSWS12 Minoru Uehara)

Tomoya Enokido was born in 1974. He received B.E. and M.E. degrees in Computers and Systems Engineering from Tokyo Denki University, Japan in 1997 and 1999. After that he worked for NTT Data Corporation, he joined Tokyo Denki University in 2002. He received his D.E. degree in Computer Science from Tokyo Denki University in 2003. He is currently a lecturer in the Faculty of Business Administration, Rissho University. His research interests include distributed systems, group communication, and distributed objects.