3 L − 6

DEADLOCK RESOLUTION IN PARALLEL EXECUTION OF NESTED TRANSACTIONS

Shinji YASUZAWA and Makoto TAKIZAWA
Tokyo Denki University

1. INTRODUCTION

A distributed system is composed of multiple objects which are interconnected by various communication networks. Objects are abstarct data types, which provide the data structure and operations for manipulating the data structure. Users issue transactions to manipulate the objects in the system. Transactions are composed of multiple operations on the objects and also atomic units of work.

Each operation a on an object o can call operations on another object. The operation a changes not only the state, i.e. data structure of o but also the states of another objects. Suppose that one operation a on an object o calls two operations b on an object p and c on q. If b and c are independent, they can be called in parallel. For each object o, some compatibility relation [KORT83] among operations is defined. In order to synchronize the executions of operations on o, some conflict-based locking mechanism is used. One problem of locking mechanism is deadlock. The deadlock problem occurred in the nested transaction is discussed in [TAKI90]. In this paper, we would like to discuss deadlock problems occurred when transactions are executed in parallel. In this paper, we assume that deadlock can be detected by some deadlock detection mechanism. If the deadlock is detected, one deadlocked transaction T is selected to be aborted to resolve the deadlock. In our system, only a part of the transaction T is aborted, i.e. partially aborted. In this paper, we present a method where some operation is executed to abort each operation to be aborted. The operation is named a *compensate* [TAKI90] of the operation.

2. SYSTEM MODEL

A distributed system M is a collection of objects which are connected by communication networks. Each object o provides data structure D_o and a set P_o of operations for manipulating D_o. An operation a can call another operations. a changes the state, i.e. both D_o and the states of objects of operations called by a. In this paper, we assume that each object o is atomic and persistent. When the object o takes an operation a from users or another object, o executes atomically a. That is, a's execution either commits or aborts. When a commits, i.e. a's execution completes successfully, a's update effect on the objects gets permanent. Even if the system suffers from some failure, the update effect by a is never lost. On the other hand, if a's execution fails, no update effect of a is reflected. Also, we assume that each object has its own recovery mechanism, i.e. log. An object o provides an operation a and its compensate operation a^{-}. The compensate operation a^{-} is defined on the basis of the application semantics.

Each object communicates with another objects to send operations and get the replies. In our system, the objects are assumed to be interconnected by a reliable broadcast communication network named an *SPO* network [NAKA89, TAKI90]. In this network, each object can broadcast messages to their destination objects in the sending order without loss.

3. TRANSACTIONS

A transaction T is composed of operations. Each operation in T can call another operations. In this sense, transactions are composed in a tree form named a transaction tree. The root denotes a transaction T named a root transaction. The leaf is named an action. Here, *[[a]]* is used to denote that an operation a is an action. The action manipulates directly an object without using any synchronization mechanism. Non-leaf nodes are called subtransactions. Each subtransaction on an object o calls actions on o and another subtransactions.

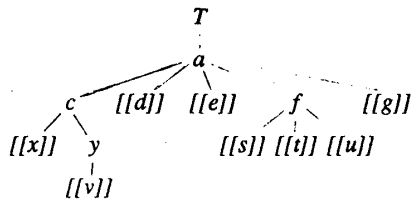

Fig.1 Transaction T

Suppose that each subtransaction a on an object o calls m operations a_1, ..., a_m. Let *[a* and *a]* denote the *begin* and *end* of a, respectively. Let Σ_a be a set $\{[a, a], a_1, ..., a_m\}$ of operations for a, i.e. operations executed in a. Σ_a is partially ordered by an ordering relation $\rightarrow_a$. $b \rightarrow_a c$ means that c is executed after b in a. If there is no ordering relation between b and c, they can be executed in parallel. Only if all the operations called by a commits, a commits. Hence, *a]* has to wait for all the operations called by a, i.e. $[a \rightarrow_a a_i \rightarrow_a a]$. For example, let us consider the operation a in Fig.1. Here, suppose that $c \rightarrow_a d$, $e \rightarrow_a f$, $e \rightarrow_a g$, and $g \rightarrow_a h$. That is, d is executed after c, f and g are executed after e, and h is executed after g. e is executed in parallel with c and d. *a]* waits for the commit of d, f, and h.

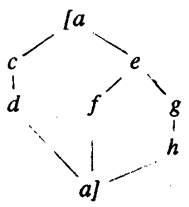

Fig.2 a's execution

DEADLOCK RESOLUTION IN PARALLEL EXECUTION OF NESTED TRANSACTIONS
Makoto TAKIZAWA and Shinji YASUZAWA
Tokyo Denki University

In this paper, locking mechanism is used as the synchronization mechanism. There are two kinds of locking schemes for nested transactions, i.e. *open* and *close*.In this paper, transactions are executed in an open scheme, because less number of locks are held while transactions are executed. Suppose that a is an operation on an object o which calls operations a_1, ..., a_m. In the open execution scheme, $[a$ means a lock operation on o, and $a]$ means unlock operations of objects obtained by a_1, ..., a_m.

4. DEADLOCK

There might occur some deadlock when the locking mechanism is used as the synchronization mechanism.

[Definition] An operation a is said to *wait on* an object o *iff* o is held by some operation b whose lock mode is incompatible with a. a is said to *depend* on an operation b ($a \Rightarrow_L b$) *iff* (1) a waits on some object held by b, (2) there is some operation c such that $a \Rightarrow_L c \Rightarrow_L b$. □

Let us consider another case in Fig.2. If T is executed in parallel, there is another deadlock situation. Suppose that $[c$ is deadlocked. Even if operations e, f, g, and h commit, they cannot release their locks until e commits, i.e. $a]$ is executed. In this sense, if some operation k in another transaction depends on f, k has to wait until c commits.

Let a and b be an operation which waits on an objec, and which holds some object in T and also is called by c. Suppose that $a \Rightarrow_L b$. If $c]$ is a lower bound of a, a waits for the release of b and b never releases the object held by b until $c]$ is executed. That is, deadlock occurs.

[Definition] Let a be an operation called by c. a is said to *implicitly depend* on b ($a \Rightarrow_M b$) *iff* (1) b is called by c and neither $a \rightarrow_T b$ nor $b \rightarrow_T c$, or (2) $[c \rightarrow_T b \rightarrow_T c]$, neither $a \rightarrow_T b$ nor $b \rightarrow_T c$, and there is some d such that $a \Rightarrow_M d$ and $d \rightarrow_T b$. □

[Definition] An operation a is said to *depend* on b ($a \Rightarrow b$) *iff* (1) $a \Rightarrow_L b$, (2) $b \rightarrow_T a$ in some transaction T, (3) $a \Rightarrow_M b$, or (4) there is some operation c such that $a \Rightarrow c \Rightarrow b$. □

Using $\Rightarrow$, the deadlock n parallel execution of transactions is defined as follows.

[Definition] An operation a is said to be *directly deadlocked iff* $a \Rightarrow a$. a is said to be *deadlocked iff* a is directly deadlocked or a depends on some deadlocked operation. □

For a system state S, let $C(T)$ be a set of operations of T which are being executed in S. The operations in $C(T)$ are said to be *current* operations of T in S. There are two kinds of current operations. The first ones are ones which wait on objects, and the others are ones which hold the objects. The formers are called *waiting* ones, and the latter ones are *executing* ones. If T is deadlocked, there exist some deadlocked waiting operations.

[Definition] A directly deadlocked operation a is said to be *waited iff* there exists an operation b in another transaction such that $a \Rightarrow b \Rightarrow a$. □

Let $W(T)$ be a set of waited operations in T if T is deadlocked. Let $LW(T)$ be a set of waited operations a such that there is no operation b in $W(T)$ such that $a \rightarrow_T b$. If the deadlock is detected and some deadlocked transaction T is detected to be aborted, all the operations in $LW(T)$ must be aborted.

5. DEADLOCK RESOLUTION

After the deadlock is detected, deadlock is resolved by aborting some transaction. In our system, compensate operations are executed for operations to be aborted. From the definition, if G^S for S includes some directed cycles, the system is deadlocked in the state S. First, one transaction T is selected in the deadlocked ones.

For T, all the operations in $LW(T)$ are aborted. Since operations which are internally called by operations in $LW(T)$ are executed or being executed, they must be stopped if they are being executed. For every operation a in $LW(T)$, a broadcasts message to every operations to stop them. If the message is received by some executing operation b for an object o_b, b's execution is stopped, i.e. aborted if $a \rightarrow_T b$. It is noted that no update effect of b remains in o_b since b does not commit. Here, let $A(T, a)$ be current operations internally called by a. Let $AO(T)$ is a set of operations internally called by operations in $LW(T)$, i.e. $AO(T) = \{ b \mid a \in LW(T)\ a \rightarrow_T b \} \cup LW(T)$. In our system, a deadlocked transaction T whose $AO(T)$ is the minimum among deadlock transactions is selected to be aborted.

The operations in $AO(T)$ are compensated by the following procedures.

[Compensate Procedure]

(1) For every operation b in $A(T, a)$, the following is done.

(2) Let c be an operation such that $c \rightarrow_T b$ and there is no operation d such that $c \rightarrow_T d \rightarrow_T b$.

(3) If c is a, a's lock is released, and then terminate.

(4) Unless every operation d such that $c \rightarrow_T d$ is compensated, it is waited.

(5) If c is an *end* $x]$, x^{-} is executed. Let b be $[x$.

(6) Otherwise, c^{-} is executed.□

For every operation a in $LW(T)$, if the compensate procedure terminates, it is re-executed.

6. CONCLUDING REMARKS

In this paper, we have discussed a deadlock in the parallel execution of nested transactions. We have presented what kind of deadlock might occur and how to resolve the deadlock in a case that each transaction is executed in parallel. In our method, some deadlocked transaction is selected and aborted partially. Also, it is aborted by executing the compensate operations. Since the compensate operation is also an operation on an object, it is executed in parallel in the same manner as the operation.

REFERENCES

[KNAP87] Knapp, E., "Deadlock Detection in Distributed Databases," ACM Computing Surveys, Vol.19, No.4, 1987, pp.303-328.

[KORT83] Korth, H. F., "Locking Primitives in a Database System," JACM, Vol.30, No.1, 1983, pp.55-79.

[NAKA89] Nakamura, A. and Takizawa, M., "Totally Ordering (TO) and Partially Ordering (PO) Broadcast Protocol," *Proc. of the 4th Joint Workshop on Computer Communication (JWCC)*, 1989, pp. 35- 43.

[MOSS85] Moss, J. E., "Nested Transactions: An Approach to Reliable Distributed Computing," The MIT Press Series in Information Systems, 1985.

[TAKI90] Takizawa, M. and Nakamura, A., "Partially Ordering Broadcast (PO) Protocol," to appear in *Proc. of the 9th IEEE Conf. on Computer Communications (Infocom)*, 1990.