Scheduling Cooperative Transactions in Distributed Database Systems

Kiyoung Kim, Jonghyun Lee, and Songchun Moon

Department of Information and Communication Engineering
Korea Advanced Institute of Science and Technology
207-43, Cheongryang, Dongdaemun, Seoul 130-012, Korea

Scheduling transactions in advanced applications of distributed database systems, such as real-time distributed groupware systems and network management systems, are different from that in traditional database systems.

1. Introduction

Transactions in advanced applications of distributed database systems, such as real-time distributed groupware systems and network management systems, have different management requirements from those in traditional applications of distributed database systems. Thus, many of the general approaches to handling concurrency in distributed database systems are inappropriate for transactions in these new types of applications. In this paper, we propose two new flexible scheduling methods for transactions in real-time distributed groupware systems and interoperable network management systems in a multi-domain network.

2. Scheduling Transactions in Real-Time Distributed Groupware Systems

Since real-time distributed groupware systems allow concurrent actions to the shared data, users will often engage in conflicting and interfering actions. This leads to a chaotic situation and must be solved by providing a scheduling algorithm for coordinating concurrent accesses to the shared data.

2.1 Limitation in Existing Scheduling Models

Transactions in real-time distributed groupware systems, different from those in conventional database systems, are (a) highly interactive and cooperative, and require (b) real-time responsiveness. Thus, its data state is generally replicated at each user’s site. Real-time responsiveness includes not only fast response time but also immediate notification, in which one user’s actions are propagated to the other users as soon as those actions occur. There may be different granularities of notification. At the finest level, each user action, e.g. keystrokes or mouse motion, is propagated. At coarser levels, user actions are grouped into larger aggregates, e.g. words or sentences, which result in notification.

A few concurrency control models have been proposed to coordinate concurrent operations in real-time distributed groupware systems: reversible execution (RE) model[1] and operation transformation (OT) model[2]. The common approach used in those models is that automatic conflict resolution is incorporated in optimistic scheduling method. That is, operations are executed immediately at local sites and then broadcast to all other sites to guarantee real-time responsiveness. If conflicts are detected at some site, an automatic conflict resolution process is performed so that the replicated data are converged quickly to the same state. For example, in RE model, a global time ordering is defined for operations, and some of conflicting operations are undone and reexecuted in the correct order. In OT model, on the other hand, conflicting operations are transformed into those that make the replicated copies be the same state before they are executed.

The main problem of hiring automatic conflict resolution in this optimistic manner is that each operation should be broadcast to all other sites, which could result in severe communication overhead. This is illustrated by Example 1.

Example 1 (Unnecessary Communication Overhead): Suppose that three authors, $U_1$ at site 1, $U_2$ at site 2, and $U_3$ at site 3, are cooperatively working on a document $D$ with a group editor. $D$ is composed of three sections, $S_1$, $S_2$, and $S_3$. $U_1$, $U_2$, and $U_3$ are currently modifying $S_1$, $S_2$, and $S_3$, respectively. While one user makes changes in his/her working data, other users see this modification which may affect their work. If $U_1$ inserts a new word composed of seven characters in $S_1$, site 1 should broadcast 7 messages to site 2 and 3, each of which depicts an insertion operation of a character at a specific position rather than broadcast single message containing the new character. That is, each finest level user actions should be broadcast to all other sites.
lating operations, based on history or type of them, is a major means of conflict resolution. This shows that optimistic scheduling methods with automatic conflict resolution, which target high responsiveness, pose severe performance problems in cases that coarser-grained notifications are preferred, due to unnecessary communication overhead. Therefore, a scheduling scheme that supports coarser-grained notifications with reasonable responsiveness is a mandatory requirement for coordinating real-time cooperative activities.

### 2.2 Light-Weight Locking

In order to support coarser level notification granularities, e.g. words or lines in text editing systems, in coordinating real-time cooperative activities, we propose a new scheduling model called **light-weight locking** (LWL). Unlike the previous approaches for coordinating cooperative activities, which pose severe communication overhead due to its adherence to the finest-level notification granularity, LWL is able to be tuned to various notification granularities without sacrificing responsiveness. This is achieved by the notions of **dynamic multi-granularity locking** and **ownership-based lock control**. They are based on the philosophy that (1) the number of locks needed in accessing data can be reduced by determining appropriate granularity of locks dynamically according to access conflicts, and (2) the communication delay in acquiring locks on replicated data can be reduced by use of **lock ownership**. Algorithm 1 shows the LWL scheduling algorithms, which are executed by the scheduler at site i.

In LWL, data items, which are replicated over all sites, are organized in a tree where small items are nested within larger ones. Any operations which updates a data should have a lock on the data or one of its ancestors which contain the data. The dynamic

#### Algorithm 1 (LWL Algorithms for Scheduling Cooperative Activities):

(a) Processing User’s Operations on Object X

**Input:** user’s operations and locking messages  
**Output:** locking messages  
**do**

1. search X’s ancestors for a locked node (LN);
2. switch (LN) {
   1. case NULL:
      1.1. owner(LN) = s_p;
      1.2. send_request();
   2. case LockS:
      2.1. execute();
   3. default: (* owner(LN) is other site *)
      3.1. if LN = X then reject o_f(X);
      3.2. else send_request();
}

**execute()**

1. MyObject = X;
2. for all o_f O(X) execute o_f;
3. for all s_i / S_i send(s_i, val(X));
4. if LockS_i = X then release LockS_i;

**send_request()**

1. if LockS_i ≠ NULL then release LockS_i;
2. send(owner(LN), LOCK_REQUEST(X));
3. await(reply);
4. switch (reply) {
   1. case LOCK_ACK: execute();
   2. case LOCK_REJECT: reject o_f(X);
}

(b) Processing Site j’s Messages Regarding Object Y

**Input:** notifications and locking messages  
**Output:** locking messages  
**do**

1. switch (m_j) {
   1.1. case val(Y): replace Y with the new version;
   2. case LOCK_REQUEST:
      2.1. search Y’s ancestors for a locked node (LN);
      2.2. switch (LN) {
         2.2.1. case LockS:
            2.2.1.1. if MyObject = Y then send(s_j, LOCK_REJECT);
            2.2.1.2. else {  
               2.2.1.2.1. Triode = SCA(MyObject, Y);
               2.2.1.2.2. LockS_i = the child of Triode which contains MyObject;
               2.2.1.2.3. LockS_j = the child of Triode which contains Y;
               2.2.1.2.4. for all s_i ∈ S_i send(s_i, LOCK_SET(LockS_i, LockS_j));
               2.2.1.2.5. send(s_j, LOCK_ACK);  
            }  
         2.2.2. case NULL:
            2.2.2.1. if s_i ≠ sp then forward m_j to %;
            2.2.2.2. else {  
               2.2.2.2.1. LockS_i = LFA(Y);
               2.2.2.2.2. for all s_i ∈ S_i send(s_i, LOCK_SET(LockS_i));
               2.2.2.2.3. send(s_j, LOCK_ACK);  
            }  
         2.2.3. default: forward mio to owner(LN);  
      }  
   3. default: (* m_j is for lock status change *)
   3.1. set the notified lock on the corresponding node; or
   3.2. release the notified lock;  
}

**Legends**

- s_p: Primary Site
- LockS_i: Site i’s Lock
- O(X): Operations on Object X
- S_i: All Sites except Site i
- send(s_p, Z): Send the message Z to Site j
- m_j: Message from Site j
- SCA(X, Y): Smallest Common Ancestor of X and Y
- LFA(X): Largest Free Ancestor of X which does not contain any locked object
multi-granularity locking aims to minimize the number of locks in accessing data without reducing concurrency. The lock granularity is determined to be the coarsest level at which users don't conflict. Coarser-grained locks are acquired first, whenever possible, and they can be dynamically re-configured to finer-grained locks before those locks are released under the conditions that a user wants to update an object of which an ancestor is locked by another user and the two users are not trying to update the same leaf node. Therefore, the dynamic multi-granularity locking supports maximum concurrency which is possible when the finest lock granularity is used, but requires fewer locks in most cases.

The ownership-based lock control aims to reduce communication delay in acquiring locks on replicated copies of the data. Distributed lock-acquiring process could be optimized by use of lock ownership. That is, the lock re-configuration process is performed only at the site which is the current owner of the lock to be re-configured. Then the re-configured locks are broadcast to all other sites.

3. Scheduling Transactions in Interoperable Network Management Systems

To provide interoperability to network management systems in a multi-domain network, we have proposed an architecture which consists of three components: data import manager, management data broker, and data export manager[3]. According to our architecture, in order to import data stored at other network management systems, management applications should send requests for data import to the management data broker (simply called "the broker" hereafter). This means that the broker should have a scheduling function to process data requests from management applications.

3.1 Problem in Optimistic Ticket Method

In our architecture, in case that a data import request is aimed at a number of data stored at local database systems(LDBSs) of different network management systems, the request inevitably becomes a global transaction that targets multiple databases. A major issue in scheduling global transactions is how to prevent inconsistent retrieval. For this, the broker should ensure serializability of global transactions, which is called global serializability.

As a way for ensuring global serializability, Georgakopoulos[4] proposed Optimistic Ticket Method(OTM) which uses tokens called tickets to determine the relative serialization order of the sub-transactions of global transactions at each LDBS. In OTM, each subtransaction of a global transaction is required to issue Take-A-Ticket operation before it submits its operations to its corresponding LDBS.

However, in case OTM is used for concurrency control of multidatabase transactions, it has a serious drawback. Since OTM allows the subtransactions of global transactions to take their tickets in any order at each LDBS, the subtransactions inherently suffer from global restarts caused by out-of-order Ticket operations between subtransactions of different global transactions. In case that a global transaction consists of a number of subtransactions which should be executed at different LDBSs, the cost of global restarts caused by out-of-order Ticket operations would be considerably significant. Example 2 illustrates how out-of-order Ticket operations between subtransactions of global transaction in OTM causes a global restart. Example 2 is originally from [5], but we have rewritten the example so as to show how the global restart caused by out-of-order Ticket operations significantly affects the execution cost of a global transaction.

Example 2 (Global Restart Problem due to Out-of-Order Ticket Operations in OTM): Consider two local database systems: LDBS1 with data items a, b, c and d, and LDBS2 with data items v, x, y and z. Let GT1 and GT2 be two read-only global transactions defined as follows.

\[
\begin{align*}
\text{GT1: } & r_1(a) r_1(b) r_1(c) r_1(v) \\
\text{GT2: } & r_2(d) r_2(x) r_2(y) r_2(z)
\end{align*}
\]

In addition, let LT1 and LT2 be two local update transactions at LDBS1 and LDBS2 defined as follows.

\[
\begin{align*}
\text{LT1: } & w_1(c) w_1(d) \\
\text{LT2: } & w_2(z) w_2(v)
\end{align*}
\]

Suppose that the ticket data items at LDBS1 and LDBS2 are denoted by \(t_1\) and \(t_2\), respectively. Finally, assume that, at LDBS1, GT1 has been executed and committed before GT2 and, at LDBS2, GT2 has been executed and committed prior to GT1. Such execution may result in the following local schedules LS1 and LS2 at LDBS1 and LDBS2, respectively.

\[
\begin{align*}
\text{LS1: } & r_1(t_1) w_1(t_1 + 1) r_1(a) r_1(b) r_1(c) c m_1 w_1(c) \\
\text{LS2: } & r_2(t_2) w_2(t_2 + 1) r_2(d) r_2(x) r_2(y) r_2(z) c m_2
\end{align*}
\]

Then, the relative serialization order of GT1 and GT2 in the graphs regarding ticket conflicts and transaction conflicts of LS1 and LS2 are as follows. Note that any local serialization graph of them does not create a cycle.

\[
\begin{align*}
\text{Ticket conflicts(LS1): } & \text{GT1} \rightarrow \text{GT2} \quad \text{Transaction conflicts(LS1): } \text{GT1} \rightarrow \text{GT2} \\
\text{Ticket conflicts(LS2): } & \text{GT2} \rightarrow \text{GT1} \quad \text{Transaction conflicts(LS2): } \text{GT2} \rightarrow \text{GT1}
\end{align*}
\]

However, the global serialization graph(GSG) covering LS1 and LS2 includes a cycle since GT1 \(\leftrightarrow\) GT2.

\[
\text{GSG(LS1): } \text{GT1} \rightarrow \text{GT2} \quad \text{GSG(LS2): } \text{GT2} \rightarrow \text{GT1}
\]

Therefore, OTM should aborts and restarts either GT1 or GT2 to disallow the non-serializable execution of their ticket operations even though they are normally completed.

3.2 Global Ticket Method
Algorithm 2 (Ensuring local serialization order):
Input: subtransaction of global transactions; Output: commit message & restart message
Begin
receive a subtransaction(ST) of a global transaction from the broker; set timeout for ST; submit ST to LDBS
wait till ∃ ST₁ such that ST₁ ∈ the set of subtransactions in prepared-to-commit state
case(ST₁)
(i) → ∃ ST₂ [global_ticket(ST₂) > global_ticket(ST₁), where ST₂ ∈ the set of uncommitted STs in the transaction table] & the timeout for ST₁ does not expire: send a commit message for ST₁ to LDBS
(ii) ∃ ST₂ [global_ticket(ST₂) > global_ticket(ST₁), where ST₂ ∈ the set of uncommitted STs in the transaction table] & the timeout for ST₁ does not expire: delay ST₁ till ST₂ is committed
(iii) the timeout for ST₁ expires: restart ST₁
end case
End

If the tickets are not maintained by LDBSs individually, global restarts aroused by out-of-order Ticket operations of subtransactions of global transactions can be avoided. Thus, we propose a scheduling method of the broker, called Global Ticket Method (GTM), that guarantees the global serializability for global transactions and eliminates global restarts incurred by out-of-order Ticket operations. We assume that all LDBSs at network management systems use two-phase locking protocol for local concurrency control and provide a prepare-to-commit state for all subtransactions of global transactions.

In order to determine the relative serialization order of global transactions at each LDBS, GTM uses global tickets that are provided by the broker. The global ticket is a timestamp whose value is stored as a data item within the broker. After the broker creates a global transaction to process an import request of a management application, the broker assigns a unique global ticket to the global transaction, and attaches the global transaction's global ticket value to subtransactions issued from the global transaction.

Subtransactions issued from a global transaction are firstly submitted to their corresponding data export managers on network management systems by the broker along with an assigned global ticket. Then, data export managers register the global ticket value and ID of the subtransactions on their transaction tables and submit the subtransactions to their LDBSs. Since each LDBS could usually receive many subtransactions belonging to different global transactions, data export managers should assure the relative serialization order among the subtransactions issued from different global transactions in the schedule of each LDBS. Algorithm 2 is to ensure the relative serialization order of subtransactions of global transactions, being executed by data export managers.

4. Conclusions

We have addressed issues of transaction scheduling in two relevant, in the sense of cooperative scheduling, target areas. For real-time distributed groupware systems, LWL supports coarser-grained notifications with reasonable responsiveness by alleviating lock and communication overhead in acquiring locks on replicated data. For interoperable network management systems, GTM reduces the communication overhead by preventing global restarts caused by out-of-order ticket operations and eliminating the interaction between the broker and network management systems for global serializability test. Further work involves the performance evaluation of LWL and GTM in various configurations.

References