Distributed Optimistic Concurrency Control for Real-Time Database Systems

Jan Lindström

Helsinki Institute for Information Technology (HIIT), Advanced Research Unit (ARU), P.O. Box 9800, FIN-02015 HUT, Finland
jan.lindstrom@cs.helsinki.fi

Abstract Concurrency control is one of the main issues in the studies of real-time database systems. In this paper different distributed concurrency control methods are studied and evaluated in a real-time system environment. Because optimistic concurrency control is a promising candidate for real-time database systems, distributed optimistic concurrency control methods are discussed in more detail. We propose a new distributed optimistic concurrency control method, demonstrate that the proposed method produces correct results and the proposed method is evaluated and tested in a prototype implementation of real-time database system for telecommunications.

1 Introduction

Numerous real-word applications contain time-constrained access to data as well as access to data that has temporal validity. As an example consider a telephone switching system, network management, navigation systems, stock trading, and command and control systems. Moreover consider the following tasks within these environments: looking up the ”800 directory”, obstacle detection and avoidance, radar tracking and recognition of objects. All of these entail gathering data from the environment, processing of information in the context of information obtained in the past, and contributing *timely* response. Another characteristic of these examples is that they contain processing both temporal data, which loses its validity after a certain time intervals, as well as historical data.

Traditional databases, hereafter referred to as databases, deal with persistent data. Transactions access this data while maintaining its consistency. The goal of transaction and query processing in databases is to get a good throughput or response time. In contrast, *real-time systems* can also deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal character of the data and the response-time requirements forced by the environment, tasks in real-time systems have time constraints, e.g., periods or deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks.

One of the most important points to remember here is that real-time does not just mean fast [SSH99]. Additionally real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time means the need to manage *explicit* time constraints in a predictable fashion, that is, to use time-cognizant protocols to deal with

deadlines or periodicity constraints associated with tasks. Databases are useful in real-time applications because they combine several features that facilitate (1) the description of data, (2) the maintenance of correctness and integrity of the data, (3) efficient access to the data, and (4) the correct executions of query and transaction execution in spite of concurrency and failures [PSRS96].

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement defined by serializability [BHG87], most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [EGLT76]. 2PL has been studied extensively in traditional database systems and is being widely used in commercial databases. In recent years, various real-time concurrency control protocols have been proposed for single-site RTDBS by modifying 2PL [HL92,AAJ92,Mar89,LHS97].

However, 2PL has some inherent problems such as the possibility of deadlocks as well as long and unpredictable blocking times. These problems appear to be serious in real-time transaction processing since real-time transactions need to meet their timing constraints, in addition to consistency requirements [Ram93].

Optimistic concurrency control protocols [KR81,Här84] have the nice properties of being non-blocking and deadlock-free. These properties make them especially attractive for real-time database systems. Because conflict resolution between the transactions is delayed until a transaction is near to its completion, there will be more information available in making the conflict resolution. Although optimistic approaches have been shown to be better than locking protocols for RTDBSs [HCL90b,HCL90a] , they have the problem of unnecessary restarts and heavy restart overhead. This is due to the late conflict detection that increases the restart overhead since some near-to-complete transactions have to be restarted. Therefore in recent years numerous optimistic concurrency control algorithms have been proposed for real-time databases [HSRT91,DVSK97,DS96,LS96,LLH95a,LLH95b].

Telecommunication is an example of an application area, which has database requirements that require a real-time database or at least time-cognizant database. A telecommunication database, especially one designed for IN services [Ahn94], must support access times less than 50 milliseconds. Most database requests are simple reads, which access few items and return some value based on the content in the database.

A main research question in this paper is, is a there distributed concurrency control method that is suitable for real-time database system. We will show that earlier methods as such are not suitable. All earlier methods require modification to support requirements set by the telecommunication environment and real-time database system. Because optimistic concurrency control is a promising candidate for real-time database systems, distributed optimistic concurrency control methods are discussed more detailed way.

This paper is organized as follows. Distributed optimistic concurrency control methods presented in literature are described and evaluated in Section 2. We will propose a new distributed optimistic concurrency control method which is presented in Section 3. Evaluation of the proposed method is presented in Section 4. Finally, Section 5 concludes this paper.

2 Distributed Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) [Här84,KR81] is based on the assumption that a conflict is rare, and that it is more efficient to allow transactions to proceed without delays to ensure serializability. When a transaction wishes to commit, a check is performed to determine whether a conflict has occurred. There are three phases to an optimistic concurrency control protocol:

- *Read phase*: The transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to a local copy of the data and announced to the database system by an operation named *pre-write*.
- *Validation phase*: The validation phase ensures that all the committed transactions have executed in a serializable fashion. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. For a transaction that contains updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained.
- *Write phase*: This follows the successful validation phase for update transactions. During the write phase, all changes made by the transaction are permanently stored into the database.

There are several ways to extend the optimistic method to the distributed case. One of the easiest is to use tickets. Others are based on optimistic locking, hybrid methods and backward validation.

We assume that each transaction receives a unique identifier at the moment it is initiated. A distributed transaction creates a subtransaction to all sites where the transaction has operations. Every subtransaction of the distributed transaction is assigned the same identifier. This is because we need to identify which transaction this subtransaction is part of.

Transaction validation is performed at two levels: local and global. The local validation level involves acceptance of each subtransaction locally. The global validation level involves acceptance of a distributed transaction on the basis of local acceptance of all subtransactions. This is similar to 2PL where the lock request in the local site relates to the local validation and 2PC relates to the global validation. Both are needed because the local serializability does not ensure the global serializability as noted in [BG81,BHG87].

2.1 Problems of Distributed Optimistic Methods

There are certain problems that arise when using optimistic concurrency methods in distributed systems [Sch82]. It is essential that the validation and the write phases are in one critical section. These operations do not need to be executed in one phase. It is sufficient to guarantee that no other validating transaction uses the same data items before an earlier validated transaction has written them.

- **Problem 1**: *Preserving the atomicity of validating and write phases* [Sch82]. One has to find a mechanism to guarantee that the validate-write critical section is atomic for global transactions, as well.

– **Problem 2**: *The validation of subtransactions is made purely on a local basis* [Sch82]. In the global validation phase, we are interested only in the order between global transactions. However, the order between distributed transactions may result from indirect conflicts, which are not visible to the global serializability mechanism. The method used also must be able to detect these indirect conflicts. These indirect conflicts are caused by local transactions which access the same data items as global transactions.
– **Problem 3**: *Conflicts that are not detectable at the validation phase*. Transaction may be non-existent in the system, active or validated. A conflict is always detected between two active transactions. Combining the phases of two transactions we can find three different combinations of states which describe different conflict detection situations.

    1. Both transactions are active during the first validation.
    2. Both transactions were active at the same time, but the conflict occurred after the first validation. This case means that remaining active transaction made a read or prewrite operation to the data item after validation of the other transaction.
    3. Transactions execute serially. Because serial execution is correct, this case is not a problem.

2.2 Optimistic Locking method

Optimistic Dummy Locking (ODL) [HD91] method combines a locking and an optimistic method. The ODL method uses dummy locks to test the validity of a transaction. The dummy locks are long-term locks; however, they do not conflict with any other locks.

In the ODL method three types of locks are used: read locks, write locks, and dummy locks. The read and write locks conflict in the usual way. A dummy lock does not conflict with any other lock. In fact, a dummy lock can be interpreted as a special mark such that it is possible to check its existence.

This method is deadlock-free, because 1) the dummy locks do not conflict with any other lock; 2) the read locks requested during the read phase are immediately released after a dummy lock is obtained; and 3) conflicting short-term read and write locks are requested in a predefined order during the validation phase.

While this method seems to be very attractive for a real-time database system, the complexity for ordering accessed data items within sites might be too heavy. It can be shown that class ODL is equivalent to the class strict 2PL [HD91]. The problem with this method is that it is based on the assumption that the serialization order of transactions is determined by the arriving order of the transactions at the validation phase. Thus the validating transaction, if not restarted, always precedes concurrently running active conflicting transactions in the serialization order. A validation process based on this assumption can incur restarts that are not necessary to ensure data consistency. These restarts should be avoided. It is not clear if this problem can be fixed to maintain the global serializability feature of the ODL.

2.3 Hybrid methods

The hybrid method based on *happened before sets* (HB) is presented in [CO82]. Every site creates a set $HB(T_i^{S_k})$ for each subtransaction $T_i^{S_k}$ at the site S_k, containing the identifiers of all subtransactions which precede $T_i^{S_k}$ in the local schedule S_k. The main idea of the global validation is the following: a subtransaction is valid if it is locally validated and all transactions which belong to its HB set have either committed or aborted. If it is not yet known for some of these transactions whether they have committed or aborted, the validation of a given subtransaction is suspended.

The above idea may be implemented in the following way. The fact that a subtransaction has been locally validated at the site is communicated to the transaction manager, which initiates and supervises transaction execution, only if for each transaction $HB(T_i^{S_k})$ the site S_k receives an acknowledgment that it has been either globally validated or aborted. Otherwise, the validation of the transaction is suspended. Suspension can produce a deadlock. In order to eliminate deadlocks, a subtransaction that cannot be validated is aborted after a predefined timeout (a deadline can be used for example). This results in the abortion of the remaining subtransactions. The timeout mechanism can obviously provoke cyclic and infinite restarting [CGM88].

Therefore this algorithm is not deadlock-free and nonblocking. This algorithm also has a priority inversion problem, because a high priority transaction might be forced to wait for lower priority transactions to complete. Therefore, this algorithm is not suitable for real-time databases.

Another hybrid method proposed is based on a modular and integrated concurrency control strategy for replicated distributed database systems [SL86]. It is modular because separate modules take care of major concurrency control functions such as management of local consistency, global consistency and mutual consistency. It is integrated because it integrates the optimistic and pessimistic approaches. The proposed algorithm allows simultaneous execution of locking transactions with optimistic transactions.

Transaction slack time in real-time databases for telecommunications are so short that in normal situations there is no time to wait for lock grants. Therefore, a pessimistic locking method is not suitable, because it contains unpredictable blocking time.

Finally, a hybrid optimistic method where phase-dependent control is utilized, such that a transaction is allowed to have multiple execution phases with different concurrency control methods in different phases is presented in [Tho98]. The proposed method uses optimistic concurrency control in the first phase and locking in the second phase. If the transaction is restarted in the first phase, then 2PL is used to limit transaction re-executions to one. This is possible, because of the very low frequency of deadlocks associated with 2PL.

This method also suffers from the need to restart a transaction that may not have enough slack time. Therefore, re-execution would be a waste of resources and would cause other transactions to miss their deadlines. Hence, this method is not possible in our environment.

3 New Distributed Optimistic Concurrency Control Method

In this section we propose a new distributed optimistic concurrency control method DOCC-DATI (Distributed Optimistic Concurrency Control with Dynamic Adjustment of the Serialization order using Timestamp Intervals). This method is based on the OCC-DATI protocol [LR99]. We have added new features to OCC-DATI to achieve distributed serializability. The commit protocol is based on 2PC, but 3PC could also be used.

We have selected object-based database as base of the telecommunication service database, because object orientation is a general trend in telecommunications standards. The presentation of the proposed method begins with a description of data objects and transactions objects in a real-time database system for telecommunication. After these essential parts of the database system have been described, we present the distributed optimistic concurrency control scheduler in detail. We present an example execution of the proposed method and demonstrate that the proposed method maintains deadlock-free, and global serializability requirements.

The concurrency control method requires certain information in order to find and resolve conflicts. This information must be gathered from the data and from the transactions. This information is read and manipulated when some transaction arrives into the system, validates or commits. Every data item in the real-time database consists of the current state of the object (i.e. current value stored in that data item), and two timestamps. These timestamps represent when this data item was last accessed by the committed transaction. These timestamps are used in the concurrency control method to ensure that the transaction reads only from committed transactions and writes after the latest committed write.

Transactions are characterized along three dimensions; the manner in which data is used by transactions, the nature of time constraints, and the significance of executing a transaction by its deadline, or more precisely, the consequence of missing specified time constraints. To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally. For simplicity of the exposition, we assume that each transaction reads and writes a data item at most once. From now on we use the abbreviations *r, w, a* and *c* for the read, write, abort, and commit operations, respectively.

A *real-time transaction* is a transaction with additional real-time attributes: deadline, priority and importance. These attributes are used by the real-time scheduling algorithm and concurrency control method.

3.1 The Proposed Method

In this section we present the basic idea of the proposed method and a sketch for its implementation. The proposed method consist of the traditional three phases of the optimistic concurrency control: read phase, validation phase, and possible write phase.

Every site contains a directory where all objects are located. Additionally, every site contains data structures for keeping transaction and object information. The transaction data structure contains information of transaction identification, the phase where a transaction is, transaction's read and write sets, and other administrative information.

These data structures are used to maintain information on the operations of the transaction and to find out what operations transaction has executed, which transactions has performed an operation on this data item and so on.

Before a transaction can enter the read phase, the transaction must be started. This is done with a **begin** operation. In the begin operation transaction is assigned a unique identifier and all other administrative information is initialized.

In the read phase if a transaction reads an object several checks must be done. Firstly, a transaction requesting the read operation must be active and not aborted. Secondly, a requested data item must not be marked as an validating object. Thirdly, if the object is not located in the local node, a **dread** operation is requested in the objects local node. A *subtransaction* is created in the receiving site which has same identity as a requesting transaction. Then we add identification of the transaction, identification of the data item and operation to the transaction read set and the data items access set. Finally, the object is copied to the transaction memory.

In the read phase if a transaction writes an object similar checks must be done using internal *prewrite* operation. Firstly, a transaction requesting the prewrite operation must be active and not aborted. Secondly, the requested data item must not be marked as an validating or preparing object. Thirdly, if the object is not located in the local node, a **dprewrite** operation is requested in the objects local node. We add identification of the transaction, identification of the data item and operation to the transaction read set and the data items access set.

In the validation phase if the transaction is a local transaction, then only local validation is executed (see Figure 2). On the other hand, if the validating transaction is a global transaction, then global validations have to be done. First, we select a coordinator to coordinate a commit protocol (2PC used here).

The coordinator will be a node where the first operation of a distributed transaction arrived. The coordinator sends a PREPARE message to all nodes where the validating transactions have operations (see Figure 1). Every participating site executes local validation (see Figure 2) and returns the result of the validation to the coordinator. In the same time, the coordinator also executes a local validation. If a validation is successful, then the participant sends a YES message to the coordinator. Otherwise the participant sends an ABORT message (see Figure 2). If all participants (coordinator included) voted YES, then the coordinator sends a COMMIT message to all participants. Otherwise the coordinator sends an ABORT message. If no vote arrives from a participant in the predefined time (deadline), then the vote is ABORT (presumed abort).

Local validation consists of iterating all objects accessed by the transaction, finding conflicting operations, and resolving conflicts (see Figure 2). The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. First we check that the validating transaction has read from committed transactions. This is done by checking the object's read and write timestamp. These values are fetched when the read and/or write to the current object is made. Then we iterate the set of active conflicting transactions. When access has been made to the same objects both in the validating transaction and in the active transaction, the temporal time interval of the active transaction is adjusted. Thus we use deferred dynamic adjustment of the serialization order.

```
occdati_commit(trid)
{
  // If transaction is distributed, use 2PC
  if ( Tr[oid].distributed == true ) {
      // send PREPARE to all participants
      for (∀ n ∈ Tr[oid].nodes ) {
        // If transaction is already aborted, then send ABORT
        if ( Tr[oid].aborted == true ) {
            send <abort,trid> to n; // Abort transaction
            return ABORT;
        }
        else send <validate,trid,node> to n; // Validate
      }
      // Collect all results starting from coordinator
      r = occdati_validate(trid);
      results = results ∪ r ;
      for (∀ n ∈ Tr[oid].nodes ) {
          r = receive(result,n);
          results = results ∪ r ;
      }
      // Check votes
      if ( ∃ r ∈ results : r == ABORT )
          for (∀ n ∈ Tr[oid].nodes ) {
              send<occdati_abort,trid>; // ABORT
              return;
          }
      else // All votes YES => COMMIT
          for (∀ n ∈ Tr[oid].nodes )
              send<occdati_fcommit,trid>;
  }
  else // Local transaction validation
      if ( occdati_validate(trid) == YES )
          return;

  occdati_fcommit(trid); // Final commit
}
```

Figure 1. Commit algorithm in the DOCC-DATI.

In local validation we need a new check for distributed objects. This is because the state of the distributed object can be changed between the last operation of the validation transaction and the validation phase by some other concurrently executing transaction. Therefore we must make sure that the transaction state is not changed. If it is, a validating transaction must be restarted. This restart could be unnecessary, but it is required to ensure distributed serializability. This new check must be done to all read-write transactions, even if the transaction is not writing to the distributed object.

This is because, the transaction is creating a new value based on an old value read from the database.

Time intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If the validating transaction is aborted no adjustments are done. A non-serializable execution is detected when the timestamp interval of an active transaction becomes empty. If the timestamp interval is empty the transaction is restarted.

```
occdati_validate(trid)
{
  if ( Tr[oid].aborted == true )  send <trid,ABORT>;

  // Select final timestamp for the transaction
  time = min(current_time, max(Tr[trid].TI));
  TI = Tr[trid].TI;
  Tr[trid].TS = time;

  // Iterate for all objects read/written
  for ( ∀ D ∈ (Tr[trid].RS ∪ Tr[trid].WS )) {
    // Data check
    if (data_check(trid, D, TI) == ABORT ) send<trid,ABORT>;

    // set prepare bookmark
    if (D ∈ TR[trid].WS )
        ReadWriteSet[oid] = ReadWriteSet[oid] ∪ < trid, prepare, 0, 0, >;

    // Calculate timestamp intervals
    adjust(trid,D,time);
  }

  // Adjust conflicting transactions
  for ( ∀ T ∈ Tr[trid].adjusted)
  {
    TI = Tr[trid].adjusted.pop(T);

    if (TI) == []) T.aborted = true;
  }

  return <trid,YES>;
}
```

Figure 2. Validation algorithm in the DOCC-DATI.

If the validating transaction has read a data item, then the validating transaction read must occur after the latest committed write to this data item (see Figure 3). If the validating transaction has announced the intention to write (prewrite) a data item, then

the validating transaction read must occur after the latest committed write and read to this data item.

```
data_check(trid,D,TI)
{
    // If this data object in validation or pre-
pared => abort now
    if (∃ l ∈ ReadWriteSet[D.oid] : l.lock == ( validate ∨ prepare) )
        return ABORT;

    // Read only from committed transactions
    if (D ∈ Tr[trid].RS )
        TI = TI ∩ [D.WTS, ∞[ ;

    if (D ∈ TR[trid].WS )
        TI = TI ∩ [D.WTS, ∞[ ∩ [D.RTS, ∞[ ;

    // Distributed objects extra check, current state check
    if (D.distributed == true and Tr[trid].readonly == false )
      TI = TI ∩ [Objects[oid].WTS, ∞[ ∩ [Objects[oid].RTS, ∞[;

    if (TI == []) return ABORT;
}
```

Figure 3. Data check algorithm in the DOCC-DATI.

If there is an active transaction which has announced the intention to write (prewrite) to the same data item which the validating transaction has read, then the active transaction write must occur after the validating transaction. Therefore, the active transaction is forward adjusted in case of a read-write conflict (see Figure 4). If there is an active transaction which has read the same data item which the validating transaction will write, then the active transactions read must be before the validating transaction. Therefore, the active transaction is backward adjusted in case of a write-read conflict. If there is an active transaction which has announced the intention to write (prewrite) to the same data item which the validating transaction will write, then the active transactions write must be after the validating transaction. Therefore, the active transaction is forward adjusted in case of a write-write conflict.

Figure 5 shows a sketch of implementing the dynamic adjustment of serialization order using timestamp intervals. This consists of two parts: forward adjustment and backward adjustment. When an active transaction has read an old value of the data item which will be replaced with a new value by the validating transaction, then the active transaction must be backward adjusted. When an active transaction writes a new value to the data item for which an old value has been read by the validating transaction, then the active transaction must be forward adjusted. This is done by removing part of

```
occdati_adjust(trid)
{
    // Conflict checking and TI calculation
    for ( ∀ T ∈ ReadWriteSet[D.oid] : t.trid != trid )
    {
      if (D ∈ (Tr[trid].RS ∩ Tr[t.trid].WS ))
        forward_adjustment(t.trid,trid,Tr[trid].adjusted,time);

      if (D ∈ (Tr[trid].WS ∩ Tr[t.trid].RS ))
        backward_adjustment(t.trid,trid,Tr[trid].adjusted,time);

      if (Di ∈ (Tr[trid].WS ∩ Tr[t.trid].WS ))
        forward_adjustment(t.trid,trid,Tr[trid].adjusted,time);
    }
}
```

Figure 4. Adjust algorithm in the DOCC-DATI.

the active transaction timestamp interval. This change is done only to temporal local variables.

If local transaction validation is successful or global transaction commit is successfully in all participant sites, then the final commit operation is executed (see Figure 6). For all objects in the validating transactions write set a validate bookmark is requested. Then the current read and write timestamps of accessed objects are updated and changes to the database are committed.

3.2 Correctness of the Proposed Method

In this section we demonstrate: First, that the DOCC-DATI method produces locally serializable histories. Secondly, that the DOCC-DATI method with the 2PC commit protocol ensures global serializability if the network is reliable. We start the demonstration by showing that if there is a conflict between two local transactions then these operations have a total order.

Lemma 31 *Let T_1 and T_2 be transactions in a history H produced by the DOCC-DATI algorithm and $SG(H)$ serialization graph. If there is an edge $T_1 \rightarrow T_2$ in $SG(H)$, then $TS(T_1) < TS(T_2)$.*

Proof: If there is an edge, $T_1 \rightarrow T_2$ in $SG(H)$, there must be one or more conflicting operations whose type is one of the following three:

1. *$r_1[x] \rightarrow w_2[x]$: This case means that T_1 commits before T_2 reaches its validation phase since $r_1[x]$ is not affected by $w_2[x]$. For $w_2[x]$, DOCC-DATI adjusts $TI(T_2)$ to follow $RTS(x)$ that is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq RTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.*

```
forward_adjustment(ttrid, trid, adjusted, time)
{
   if (ttrid ∈ adjusted)
     TI = adjusted.pop(ttrid);
   else
     TI = Tr[ttrid].TI;

   TI = TI ∩ [time + 1, ∞[ ;
   adjusted.push({(ttrid, TI)});
}

backward_adjustment(ttrid, trid, adjusted, time)
{
   if (ttrid ∈ adjusted)
     TI = adjusted.pop(ttrid);
   else
     TI = Tr[ttrid].TI;

   TI = TI ∩ [0, time − 1] ;
   adjusted.push({(ttrid, TI)});
}
```

Figure 5. Backward and Forward adjustment in the DOCC-DATI.

2. $w_1[x] \rightarrow r_2[x]$: *This case means that the write phase of T_1 finishes before $r_2[x]$ executes in T_2's read phase. For $r_2[x]$, DOCC-DATI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.*
3. $w_1[x] \rightarrow w_2[x]$: *This case means that the write phase of T_1 finishes before $w_2[x]$ executes in T_2's write phase. For $w_2[x]$, DOCC-DATI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.* □

Using Lemma 31 we can now prove that the local history produced by the DOCC-DATI scheduler is serializable. This is done by means of contradiction.

Theorem 31 *Every local history generated by the DOCC-DATI algorithm is serializable.*

Proof: Let H denote any history generated by the DOCC-DATI algorithm and $SG(H)$ its serialization graph. Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 1, we have $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. By induction we have $TS(T_1) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in $SG(H)$ and thus the DOCC-DATI algorithm produces only serializable histories. □

Because local serializability does not ensure global serializability we have to provide some demonstration that the proposed DOCC-DATI will also ensure global seri-

```
occdati_fcommit(trid)
{
  time = Tr[trid].TS; /* Final commit time */

  // Update object timestamps
  for ( ∀ D ∈ (Tr[trid].RS ∪ Tr[trid].WS ))
  {
    // set validate bookmark
    if (D ∈ TR[trid].WS )
        ReadWriteSet[oid] = ReadWriteSet[oid] ∪ <trid, validate, 0, 0, >;

    if (D ∈ Tr[trid].RS )
      Objects[D.oid].RTS = max(Objects[D.oid].RTS, time);
    if (D ∈ WS(T_v))
      Objects[D.oid].WTS = max(Objects[D.oid].WTS, time);
  }

  commit T_v to database; // Write phase
}
```

Figure 6. Final commit algorithm in DOCC-DATI.

alizability. First, if some validating transaction is marked a data item by a validation mark, then no other validating or active transaction can access the same object (these transactions were restarted). Because validation is done in a critical section in the site, there can only be one validation ongoing in one site. If two transactions enter the validation phase simultaneously in different sites and both access the same data item, then at least one of the validating transactions is aborted. In the worst case, both of them can be aborted if both have used a common data item in both sites.

When a validating transaction has acquired all validation marks for all objects in its write set on all sites, where the validating transaction has operations, no other transaction can access those objects. Because 2PC decides the distributed commit problem in case of reliable communication network, a validating transaction eventually decides to commit or abort in all sites and the decision is the same on all sites.

Theorem 32 *Every global history generated by the DOCC-DATI algorithm is serializable.*

Proof: Let GH denote any global history generated by the DOCC-DATI algorithm and $SG(GH)$ its global serialization graph. Suppose, by way of contradiction, that $SG(GH)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$, where $n > 1$.

Proof is done by induction on n. The basic step, for $n = 2$, there are two two possible cases:

- *Case 1: Both transactions are local transactions. Then by Theorem 31 we have $TS(T_1) < TS(T_1)$. This is a contradiction and therefore no cycle can exist.*

– *Case 2: Both transactions are global transactions. Because the proposed method executes global operations in both sites, thus the serialization graph for both sites are identical. Therefore, by Theorem 31 we have* $TS(T_1) < TS(T_1)$. *This is a contradiction and therefore no cycle can exist.*

Suppose that the Theorem holds for $n = k$ *for some* $k \geq 2$. *We will show that it holds for* $N = k + 1$. *By the induction hypothesis, the path* $T_1 \rightarrow ... \rightarrow T_k$ *implies that* $TS(T_1) < TS(T_k)$. *By* $T_k \rightarrow T_{k+1}$ *we can have the following two possible cases:*

– *Case 1: Both transactions* T_k *and* T_{k+1} *are local transactions. Then by Lemma 31 we have* $TS(T_1) < TS(T_k) < TS(T_{k+1}) < TS(T_1)$. *This is a contradiction and therefore no cycle can exists.*
– *Case 2: Both transactions* T_k *and* T_{k+1} *are global transactions. Then by Lemma 31 we have* $TS(T_1) < TS(T_k) < TS(T_{k+1}) < TS(T_1)$ *at least one site. This is a contradiction and therefore no cycle can exists.*

There is at least one transaction which must be forward adjusted and at least one transaction which must be backward adjusted if we have a global deadlock. Let us assume that one transaction from the cycle enters the validation phase. Let us denote backward adjusted transactions final timestamp by $TS(T_b)$. *Because there is a cycle in the global serialization graph, the backward adjusted transaction is writing some data item. Thus, the backward adjusted transaction is not a read-only transaction. Therefore, an extra check must done in this transaction for the distributed object.*

If there is a cycle, there must be at least two objects which at least two distributed transactions (denote them by T_{g_1} *and* T_{g_2} *) have written. The backward adjusted transaction is read one of these distributed objects (lets assume that it is* X*). This case means that* T_b *commits before* T_{g_1} *reaches its validation phase since* $r_b[X]$ *is not affected by* $w_{g_1}[X]$. *For* $w_{g_1}[X]$, *DOCC-DATI adjusts* $TI(T_{g_1})$ *to follow* $RTS(X)$ *that is equal to or greater than* $TS(T_b)$. *Thus,* $TS(T_b) \leq RTS(X) < TS(T_{g_1})$. *Therefore,* $TS(T_b) < TS(T_{g_1})$.

Similarly, the same backward adjusted transaction has written some new value (because there is a cycle, let us assume that this data item is Y, *note that* $X = Y$ *is possible). Because there is a cycle, at least one global transaction has been read the old value of the* Y *(let us assume that it is* T_{g_2}*). This case means that* T_b *commits before* T_{g_2} *reaches its validation phase since* $r_{g_2}[Y]$ *is not affected by* $w_b[Y]$. *For* $r_{g_2}[Y]$, *DOCC-DATI adjusts* $TI(T_{g_2})$ *to follow* $WTS(Y)$ *that is less or equal than* $TS(T_b)$. *Thus,* $TS(T_{g_2}) \leq WTS(Y) < TS(T_b)$. *Therefore,* $TS(T_{g_2}) < TS(T_b)$.

Therefore, by Lemma 1, we have $TS(T_1) < TS(T_2) < TS(T_{g_2}) < TS(T_b) < TS(T_{g_1}) < ... < TS(T_n) < TS(T_1)$. *Therefore, by induction we have* $TS(T_1) < TS(T_1)$. *This is a contradiction. Therefore no cycle can exist in* $SG(GH)$ *and thus the DOCC-DATI algorithm produces only globally serializable histories.* □

Proposed method is deadlock-free, because there is no waiting in any request of operation.

3.3 Working example

In this section we provide an example execution of the proposed method. We assume a two-node full network. In the example history we have three transactions (see Figure 7):

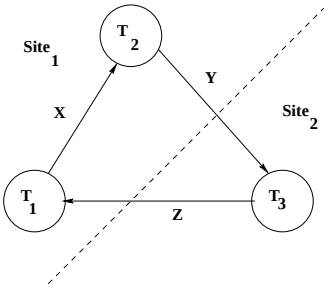


Figure 7. Database in the example.

Transactions and history is following:

$T_1 : r_1[X]r_1[Z]w_1[Z]$

$T_2 : r_2[Y]r_2[X]w_2[X]$

$T_3 : r_3[Z]r_3[Y]w_3[Y]$

$H : r_1[X]r_2[Y]r_3[Z]r_1[Z]r_2[X]r_3[Y]w_2[X]c_2w_1[Z]w_3[Y]c_1c_2$

Objects X and Y are stored in node $site_1$ and object Z in $site_2$. Transactions T_1 and T_2 start in node $site_1$ and transaction T_2 in $site_2$. Transaction T_1 and T_3 are distributed transactions and transaction T_2 is a local transaction. Let us assume that $RTS(X, Y, Z) = WTS(X, Y, Z) = 0$. Now let us execute the history.

1. $r_1[X]r_2[Y]r_3[Z]$: local read operations. Set transaction identification, object identification and operation to transactions read set and objects access set. These operations can be executed in parallel.
2. $r_1[Z]$: global read operation. Send operation to $site_2$. Set transaction identification, object identification and operation to transactions read set and objects access set. Send object to requesting site (i.e site $site_1$). Set transaction identification, object identification and operation to transactions read set and objects access set. Therefore, bookmark $(T_1) \xrightarrow{r} Z$ is in site $site_1$ and bookmark $(T_3, T_1) \xrightarrow{r} Z$ is in site $site_2$.
3. $r_2[X]$: local read operation. Set transaction identification, object identification and operation to transactions read set and objects access set. Therefore, bookmark $(T_1, T_2) \xrightarrow{r} X$ is in sites $site_1$ and $site_2$. This operation can be executed concurrently (i.e. interleaved) with the above operation.
4. $r_3[Y]$: global read operation. Send operation to $site_1$. Set transaction identification, object identification and operation to transactions read set and objects access set. Send object to requesting site (i.e site $site_2$). Set transaction identification, object

identification and operation to transactions read set and objects access set. Therefore, bookmark $(T_2) \xrightarrow{r} Y$ is in site $site_1$ and bookmark $(T_2, T_3) \xrightarrow{r} Y$ is in site $site_2$.

5. $w_2[X]$:local write operation. Set transaction identification, object identification and operation to transactions write set and objects access set. Therefore, bookmark $(T_1) \xrightarrow{r} X$ and bookmark $(T_2) \xrightarrow{rw} X$ is in site $site_1$ This operation could be executed concurrently with the above operation.
6. c_2: validation of the local transaction. First, the final timestamp for transaction T_2 is selected. Because transaction T_2 is not adjusted, thus final timestamp is the current time (i.e. $TS(T_2) = current_time$). Transaction T_1 has read an old value of the data item X. Therefore, transaction T_2 is backward adjusted to be $TI(T_1) = [0, TS(T_2)-1[$. The write timestamp of the data item X is updated, thus $WTS(X) = RTS(X) = TS(T_2)$.
7. $w_1[Z]$: global write operation. Send operation to $site_2$. Set transaction identification, object identification and operation to transactions write set and objects access set. Send object to requesting site (i.e site $site_1$). Set transaction identification, object identification and operation to transactions write set and objects access set. Therefore, bookmark $(T_1) \xrightarrow{rw} Z$ is in site $site_1$ and bookmark $(T_1) \xrightarrow{rw} Z$ and bookmark $(T_3) \xrightarrow{r} Z$ are in the site $site_2$.
8. $w_3[Y]$: global write operation. Send operation to $site_1$. Set transaction identification, object identification and operation to transactions write set and objects access set. Send object to requesting site (i.e site $site_2$). Set transaction identification, object identification and operation to transactions write set and objects access set. Therefore, bookmark $(T_3) \xrightarrow{rw} Y$ is in site $site_2$ and bookmark $(T_3) \xrightarrow{rw} Y$ is the site $site_1$. This operation could be executed concurrently with the above operation.
9. c_1: validation of the global transaction. The coordinator is the node $site_1$. A validation request is sent to the nodes $site_1$ and $site_2$. In $site_1$ validation succeed, because there are no active conflicting transactions. In $site_2$, there is an active conflicting transaction T_3 which has read the old value of the data item Z. Therefore, the timestamp interval of transaction T_3 is backward adjusted to be $TS(T_3) = [0, TS(T_1) - 1]$. Finally, the timestamps of the data item Z are updated to be $WTS(Z) = RTS(Z) = TS(T_1)$.
10. C_3: validation of the global transaction. The coordinator is the node $site_2$. The validation request is sent to the nodes $site_1$ and $site_2$. In $site_1$ validation succeed, because there are no active conflicting transactions. In $site_2$, there are no active conflicting transactions, but a transaction has been read an old value of the distributed object Z. Because the transaction is not read-only, an extra check is done to make sure that the decision is based on the current state of the database. Therefore, the transaction timestamp interval is forward adjusted to be $TS(T_3) = [0, TS(T_1) - 1] \cap [WTS(Z) = TS(T_1), \infty[= \emptyset$. The transactions timestamp interval is empty, thus the transaction is restarted.

4 Evaluation

The prototype system used in the evaluations is based on the *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) [LNPR99] specification.

RODAIN Database Nodes that form one RODAIN Database Cluster are real-time, highly-available, main-memory database servers. They support concurrently running real-time transactions using an optimistic concurrency control protocol with deferred write policy. They can also execute non-real-time transactions at the same time on the database. Real-time transactions are scheduled based on their type, priority, mission criticality, or time criticality. All data in the database is stored in the main-memory database. Data modification operations are logged to the disk for persistence.

In order to increase the availability of the database each Rodain Database Node consists of two identical co-operative units. One of the units acts as the Database Primary Unit and the other one, Database Mirror Unit , is mirroring the Primary Unit. Whenever necessary, i.e. when a failure occurs, the Primary and the Mirror Units can switch their roles.

The database server was running on an Intel Pentium 450 MHz processor with 256 MB of main memory. A similar computer was used for the client. The computers were connected using a dedicated network, the speed of which was controlled by changing the hub connecting the computers. To avoid unnecessary collisions, there was no other network traffic while the measurements were performed.

Database used in tests is based on a GSM model and transactions are simple transactions accessing Home Location Register (HLR) and Visitor Location Register (VLR). The database size is 30000 items. The transactions used and their ratios are presented in table 1.

Table 1. Transactions used in the evaluation.

Transaction	type	ratio
GetSubscriber (HLR)	local read	70 %
GetSubscriber (VLR)	remote read	20 %
UpdateSubscriber	remote write	10 %

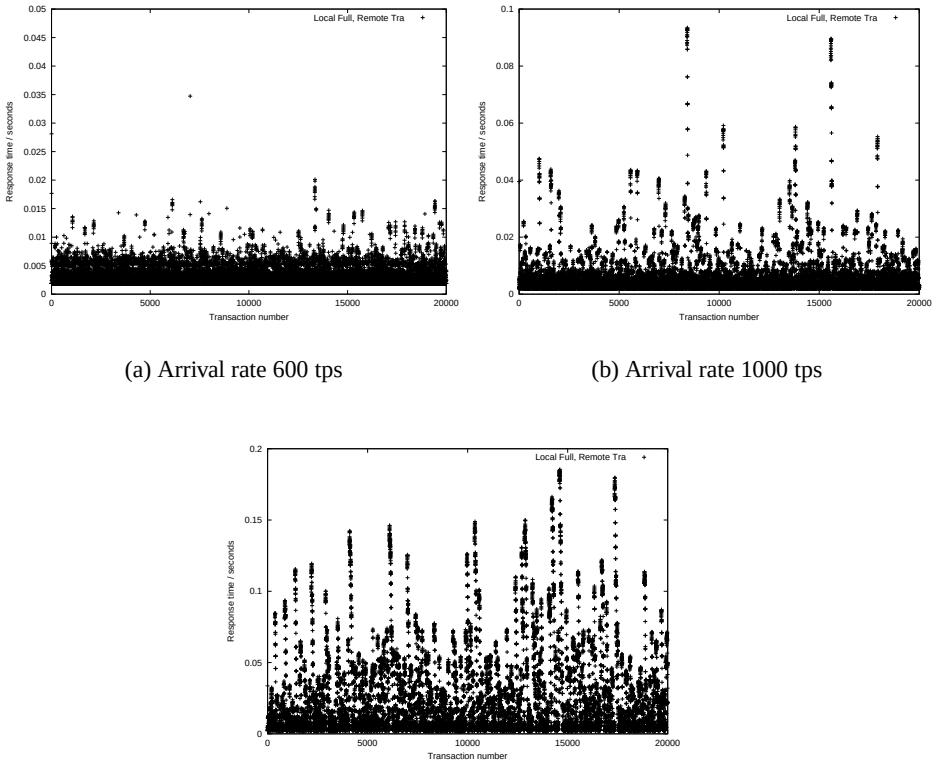
All time measurements were performed on the client computer using the gettimeofday function, which provides the time in microseconds. The client sends the requests following a given plan, which describes the request type and the time when the request is to be sent. When the request is about to be sent the current time is collected and when the reply arrives the time difference is calculated.

Linux provides static priorities for time-critical applications. These are always scheduled before the normal time-sharing applications. The scheduling policy chosen was Round-robin (SCHED_RR) using the scheduler function sched_setscheduler.

The database was also avoiding swapping by locking all the process pages in the memory using mlockall function. The swap causes long unpredictable delays, because occasionally some pages are sent and retrieved from the disk. Because in our experiment environment our database system was the only application running no swapping occurred during the tests.

With a low arrival rate the system can serve all requests within the deadlines. The single highest response time with 600 transactions per second is nearly 35 milliseconds (see Figure 8(a). A moderate arrival rate, 1000 tps (see figure 8(b)), creates occasional

situations, when the response time temporarily is higher than the deadline. The transaction are treated similar in the measurements, because the service sequence does threat the differently. In the overload situation (arrival rate 1600 tps, see Figure 8(c)), the system is capable of serving most requests still within the deadline. Unfortunately, there is no trend to predict which requests are served fast enough. Only a bit less than 20% (3400 requests out of the studies 20000) of all requests have response times over the deadline. All kinds of requests belong to this 'over the deadline' group. The ratios of the served requests in this group are similar to the ratios of the original requests in the whole set.



(a) Arrival rate 600 tps

(b) Arrival rate 1000 tps

(c) Arrival rate 1600 tps

Figure 8. Two database nodes. Local using Primary, Mirror and SSS units. Remote using only Transient unit

The comparison of the arrival rates is shown in table 2. From these values we can see that the database can provide suitable response time to the requests even when the occasional delays are longer than the allowed deadline 50 ms.

Table 2. Statistics from benchmark evaluation, where only the arrival rate was varied.

Arr. rate	Min	Max	Median	Average	Deviation
600 tps	0.0019	0.0347	0.0029	0.0034	0.0018
1000 tps	0.0019	0.0934	0.0037	0.0060	0.0076
1200 tps	0.0019	0.0824	0.0046	0.0082	0.0096
1600 tps	0.0019	0.1855	0.0139	0.0267	0.0313

5 Conclusions

In this paper we have reviewed different distributed concurrency control techniques and their complexity. The study has focused on distributed optimistic concurrency control methods, because optimistic concurrency control has been shown to be applicable to real-time database systems.

Our study has shown that there is no distributed optimistic concurrency control method that is clearly suitable for a real-time database system for the telecommunication environment without modifications. Therefore, we have presented and evaluated a new distributed optimistic concurrency control method DOCC-DATI. We have presented the basic idea of the proposed method with prototype implementation. Additionally, we have demonstrated that the proposed method is deadlock-free, locally non-blocking, and produces only locally and globally serializable histories. Therefore, the proposed method is good candidate for a concurrency control method for distributed real-time database systems.

The proposed method should be evaluated with some well known and widely used method. Therefore, we have selected 2PL-HP as reference method. With DOCC-DATI we will use the 2PC commit protocol and with 2PL-HP method. We will implement a prototype system or simulation model for testing these methods in practice.

References

[AAJ92] D. Agrawal, A. E. Abbadi, and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 104–113. ACM Press, 1992.

[Ahn94] I. Ahn. Database issues in telecommunications network management. *ACM SIGMOD Record*, 23(2):37–43, June 1994.

[BG81] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surverys*, 13(2), June 1981.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[CGM88] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency Control in Distributed Database Systems*. Addison-Wesley, 1988.

[CO82] S. Ceri and S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–130, 1982.

[DS96] A. Datta and S. H. Son. A study of concurrency control in real-time active database systems. Tech. report, Department of MIS, University of Arizona, Tucson, 1996.

[DVSK97] A. Datta, I. R. Viguier, S. H Son, and V. Kumar. A study of priority cognizance in conflict resolution for firm real time database systems. In *Proceedings of the Second International Workshop on Real-Time Databases: Issues and Applications*, pages 167–180. Kluwer Academic Publishers, 1997.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[H¨ar84] T. H¨arder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[HCL90a] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 94–103. IEEE Computer Society Press, 1990.

[HCL90b] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM Press, 1990.

[HD91] U. Halici and A. Dogac. An optimistic locking technique for concurrency control in distributed databases. *IEEE Transactions on Software Engineering*, 17(7):712–724, July 1991.

[HL92] S.-L. Hung and K.-Y. Lam. Locking protocols for concurrency control in real-time database systems. *ACM SIGMOD Record*, 21(4):22–27, December 1992.

[HSRT91] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th VLDB Conference*, pages 35–46, Barcelona, Catalonia, Spain, September 1991. Morgan Kaufmann.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[LHS97] K.-Y. Lam, S.-L. Hung, and S. H. Son. On using real-time static locking protocols for distributed real-time databases. *The Journal of Real-Time Systems*, 13(2):141–166, September 1997.

[LLH95a] K.-W. Lam, K.-Y. Lam, and S. Hung. An efficient real-time optimistic concurrency control protocol. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 209–225. Springer, 1995.

[LLH95b] K.-W. Lam, K.-Y. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, pages 174–179. IEEE Computer Society Press, 1995.

[LNPR99] J. Lindstr¨om, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*, Lecture Notes in Computer Science, vol 1819, pages 158–173, 1999.

[LR99] J. Lindstr¨om and K. Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time databases. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 13–20. IEEE Computer Society Press, 1999.

[LS96] J. Lee and S. H. Son. Performance of concurrency control algorithms for real-time database systems. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 429–460. Prentice-Hall, 1996.

[Mar89] K. Marzullo. Concurrency control for transactions with priorities. Tech. Report TR 89-996, Department of Computer Science, Cornell University, Ithaca, NY, May 1989.

[PSRS96] B. Purimetla, R. M. Sivasankaran, K. Ramamritham, and J. A. Stankovic. Real-time databases: Issues and applications. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 487–507. Prentice Hall, 1996.

[Ram93] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, April 1993.

[Sch82] G. Schlageter. Problems of optimistic concurrency control in distributed database systems. *ACM SIGMOD Record*, 13(3):62–66, April 1982.

[SL86] A. P. Sheth and M. T. Liu. Integrated locking and optimistic concurrency control in distributed database systems. In *Proceedings of the 6th Int. Conf. on Distributed Computing Systems*, pages 89–99, 1986.

[SSH99] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.

[Tho98] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):173–189, 1998.