The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource-Managers

Yoav Raz

EMC Corporation

Technical Report

November 16, 1990
Revised April 23, 1995
Version 1.9

Preface

In the beginning of 1990 S-S2PL was believed to be the most general history property that guarantees global serializability over autonomous RMs. The discovery of Commitment Ordering (CO) resulted from an attempt to generalize S-S2PL, i.e., to find a superset of S-S2PL that guarantees global serializability while allowing to maintain RM autonomy. An intermediate step was discovering a property that I named *Separation* (SEP; "separating" conflicting operations by a commit event), which is a special case of CO. Separation is defined as follows (compare with CO):

A history H is in SEP (is *separated*) if for any conflicting operations $p_1[x]$, $q_2[x]$ of any *committed* transactions T_1, T_2 respectively in H, $p_1[x] < q_2[x]$ implies $e_1 < q_2[x]$.

Note that the commit projection of a history in SEP is in S-S2PL.

Separation can be enforced by using algorithm 4.1 (the CO algorithm), where the set $ABORT_{CO}(T)$ is replaced with $ABORT_{SEP}(T) = \{ T' \mid T' \rightarrow T$ or $T \rightarrow T'$ is an edge in the USG $\}$
(i.e., aborting both "forwards" and "backwards", while CO only requires aborting "backwards").

SEP is less restrictive than S-S2PL, and allows optimistic implementations. However, it is far more restrictive than CO. It was noticed later that the *Optimistic 2PL* scheduler described in [Bern 87] spans exactly the SEP class.

A paper on separation, similar to this one, was written by me in July-August 1990, and included results for SEP, parallel to most main results for CO, described here. The first version of the current CO paper was a rewrite of the separation paper. The separation paper included an erronous theorem, claiming that SEP was the most general property (a necessary condition for) guaranteeing global serializability over autonomous RMs. The proof was almost identical to the proof of theorem 6.2 here. CO was formulated two days after I noticed a mistake in the proof of that theorem: SEP requires aborting more transactions than the minimum necessary to guarantee global serializability. This minimum is exactly defined by $ABORT_{CO}(T)$, when T is committed. *Extended CO* (ECO; [Raz 91b] was formulated a few days later.

Y. R.

Revisions:

Version 1.0	November 16, 1990	Digital Equipment Corporation
Version 1.1	June 21, 1991	
Version 1.2	April 23, 1992	
Version 1.3	May 18, 1992	
Version 1.4	June 2, 1992	
Version 1.5	June 15, 1993	
Version 1.6	July 2, 1993	Appendix added.
Version 1.7	December 15, 1994	EMC Corporation
Version 1.8	December 30, 1994	
Version 1.9	April 23, 1995	

Contents

1 Introduction 2

2 Histories and their properties - an overview 7

2.1 Transactions and histories 7

2.2 On guaranteeing a property 8

2.3 History classes 9

2.3.1 Serializability 9

2.3.2 Recoverability 11

2.3.3 Two Phase Locking 12

2.4 On inherently-blocking and noninherently-blocking properties 12

2.5 On commit-decision delegation 13

3 Commitment Ordering (CO) 15

3.1 CO as a serializability concept 15

3.2 CO - recoverable histories 17

3.3 Relationships among properties - a summary 18

4 Commitment Ordering schedulers 20

4.1 Schedulers: components and classification 20

4.2 A "pure" CO TTS - The Commitment Order Coordinator (COCO) 20

4.3 The CO-Recoverability Coordinator (CORCO) 23

4.4 Combining a CO TTS with a RM's scheduler 26

4.5 Locking based Strict CO (SCO) 28

4.6 Timestamp based CO scheduling 30

4.7 Multi-version based CO scheduling 32

5 Multiple Resource Manager environment 37

5.1 The underlying transaction model 37

5.2 Local and global properties 40

5.3 On generating global CO histories: The distributed CO algorithm 43

5.4 On CO scheduling and global deadlocks 45

6 Guaranteeing global serializability by local Commitment Ordering 47

6.1 Local-CO is sufficient for global serializability 47

6.2 Conditions when Local-CO is necessary to guarantee global serializability . 47
7 Conclusion 51
Acknowledgments 52
References 52
Appendix - Commitment Ordering Architecture 56
A.1 Common Architecture for Transaction Management 56
A.2 The Commitment Order Coordinator (COCO) of a Resource Manager 59
A.2.1 Interfaces 61
A.2.2 The CO-AC Algorithm 63
A.2.2.1 Invocation sequences 63
A.2.2.2 The COCO's invocations 65
A.2.2.3 The VOTE algorithm 65
A.3 Extensions of the COCO 66
A.3.1 Distingushing between local and global transactions 66
A.3.2 Distingushing between queries and updaters 67
A.3.3 A COCO that enforces recoverability (CORCO) 67
A.4 Summary 68

The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource-Managers

Yoav Raz[1]

Abstract

Commitment Ordering (CO) is a serializability concept that allows *global serializability* to be effectively achieved across multiple *autonomous Resource Managers* (RMs). The RMs may use different (any) concurrency control mechanisms. Thus, CO provides a solution for the long standing global serializability problem.

RM *autonomy* means that no concurrency control information is shared with other entities, except that of *Atomic Commitment* (AC) protocol messages (e.g., *Two Phase Commitment - 2PC*). CO is a necessary and sufficient condition for guaranteeing global serializability across autonomous RMs. CO guarantees global serializability by utilizing AC protocols to achieve consensus among participants of a distributed transaction on both *atomicity* ("all or nothing" semantics) and serializability. Thus no communication overhead is incurred by CO.

CO generalizes the well known *Strong-Strict Two Phase Locking* concept (S-S2PL; "release locks applied on behalf of a transaction only after the transaction has ended"). While S-S2PL is subject to *deadlocks*, CO exhibits local *deadlock-free* executions when implemented as *non-blocking* (optimistic) concurrency control mechanisms.

Recovery procedures available by the individual RMs, as well as by the AC protocols used, are unaffected by enforcing CO.

[1] EMC Corporation, 171 South Street, Hopkinton, Ma 01748. (508) 435 1000 ext. 4146, raz@emc.com

1 Introduction

Distributed transaction management services are intended to provide coordination for transactions that span multiple *resource managers* (RMs).

An RM is a software component that manages resources under transactions' control. A *resource* is any entity/object with well defined *states* that are being modified and retrieved while obeying transaction's ("all or nothing") semantics (*atomicity*). This means that effects of failed transactions are undone, which requires that resources' states be *recoverable* (i.e., if a resource is modified by a transaction, the state it had when the transaction started can be restored before the transaction ends*).* A resource is typically (but not necessarily) a data item. The scope of any specific resource (e.g., granularity units, versions, or replications) is defined as a part of an RM's semantics. Examples of resource managers are database systems (DBSs), queue managers, cache managers, certain types of management objects (e.g., [OSI-SMO]) etc.

An RM may impose a certain property of the generated transaction *histories* (transaction event schedules) to guarantee correctness and certain levels of fault tolerance. However, the *global history,* i.e., the combined history of all the RMs involved, does not necessarily inherit such a property even if it is provided by all the RMs. The *serializability* (SER) property is an example. Serializability is the most commonly accepted general criterion for the correctness of concurrent transactions (e.g., see [Bern 87], [Papa 86]), and supported in most RMs. When transactions involve more than one RM, this property may be violated in general, unless special measures are taken, or certain conditions exist to guarantee it. The problem, as well as the importance of guaranteeing global serializability are demonstrated in example 1.1:

Example 1.1

Two transaction programs execute over two different bank DBSs. The transaction programs and the DBSs may execute on (four) different nodes in a network.

- Transaction T1 transfers $100 from account A @ bank AA (DBS AA) to account B @ bank BB (DBS BB).
  The related application program and database access operations are as follows:

Program steps	**Corresponding database operations**
begin T_1	
read (A@AA)	$r_1[A]$
A:=A - 100	
write(A@AA)	$w_1[A]$
read (B@BB)	$r_1[B]$
B := B + 100	
write(B@BB)	$w_1[B]$
end T_1	

- Transaction T2 retrieves and displays the balances sum of accounts A and B:

Program steps	**Corresponding database operations**
begin T_2	
read (B@BB)	$r_2[B]$
read (A@AA)	$r_2[A]$
display(A + B)	
end T_2	

Suppose that two users invoke T_1 and T_2 independently, concurrently. One of the possibilities for the DBSs scheduling operations implied by T_1 and T_2 is as follows:

DBS AA: ...$r_1[A=1000]$ $w_1[A=900]$ c_1 $r_2[A=900]$ c_2

DBS BB: ...$r_2[B=2000]$ $r_1[B=2000]$ $w_1[B=2100]$ c_1 c_2

where c_1 and c_2 are the commit events of T_1 and T_2, respectively.

Each DBS's schedule is serializable, but the combined (global) schedule is not. Due to global serializabiliry violation, the result displayed by T_2, \$2900, is erroneous (should be \$3000). Note that the common locking scheme (S-S2PL; see more below) results for this scenario in a global deadlock, which prevents the error.

§

The problem of global serializability is dealt with, for example, in [Brei 90], [Brei 91], [Brei 92], [Elma 87], [Geor 91], [Glig 85], [Litw 89], [Papa 86], [Pu 88] and [Veij 92]. [Weih 89] deals with the relationships between local and global serializability in the framework of abstract data types. Achieving global serializability with reasonable performance, especially across RMs that implement different concurrency control mechanisms, has been considered a difficult problem (e.g., [Papa 86], [Shet 90]), and recently described as *open* ([Silb 91]):

- C. Papadimitriou ([Papa 86], page 223):
  "In conclusion, distributed concurrency control algorithms exhibits the following very intricate behavior with respect to performance: At an excessive communication cost, we can always reduce the distributed concurrency control problem to the centralized one, and thus realize by a time-efficient scheduler any concurrency control principle which has polynomially recognizable prefixes. If communication costs are a matter of concern, then we can realize any concurrency control principle using the absolutely minimum number of messages possible in polynomial space. Can we realize such a concurrency control principle at the minimum com-

munication cost in polynomial time? The next result says that the answer is negative even for simple and practically useful concurrency control principles as serializability, unless NP = PSPACE[1]".

- A. Sheth and J. Larson ([Shet 90], page 227):
  "Without knowledge about local as well as global transactions, it is highly unlikely that efficient global concurrency control can be provided... Additional complications occur when different component DBMSs and the FDBMS[2] support different concurrency mechanisms... It is unlikely that a theoretically elegant solution that provide conflict serializability without sacrificing performance (i.e., concurrency and/or response time) and availability exists".

- A. Silberschatz, M. Stonebraker, and J. Ullman ([Silb 91], page 120):
  Transaction management in a heterogeneous, distributed database system is a difficult issue. The main problem is that each of the local database management systems may be using a different type of cuncurrency control scheme. Integrating this is a challenging problem, made worse if we wish to preserve the local autonomy of each of the local databases, and allow local and global transactions to execute in parallel. One simple solution is to restrict global transactions to retrieve-only access. However, the issue of reliable transaction management in the general case, where global and local transactions are allowed to both read and write data, is still open".

Global serializability can be guaranteed, in principle, by several methods, if the RMs involved share relevant concurrency control information. *Timestamp Ordering* (*TO*) is an example (e.g., [Bern 87], [Lome 90]). If all the RMs involved support TO-based concurrency control and share the same timestamps, then the entire system can exhibit a coherent behavior based on TO, which guarantees global serializability. However, this technology requires a certain RM synchronization as well as timestamp propagation, and is currently unavailable in heterogeneous environments. Generating timestamps in a distributed environment, that match conflicting operations orders, while minimizing resulting aborts (due to mismatch) is not trivial (e.g., see [Lomet 90]).

Another known method, based on *locking*, allows RM *autonomy*. We define autonomy as follows:

Definition 1.1

An RM is *autonomous*, if it does not have to share any resources and concurrency control information (e.g., timestamps) with another entity (external to the RM), and is being coordinated (at the nonapplication level[3]) solely via *Atomic Commitment* (AC) protocols.

§

Since the discussion here is confined to atomic transactions only, AC protocols (for enforcing global atomicity; e.g., see [Bern]) are necessary in a multi RM environment, and thus, the definition of RM *autonomy* implies a minimal requirement for RMs' cooperation. Most systems that support distributed transaction services provide AC protocols and related interfaces. These protocols guarantee atomicity even in the presence of certain types of recov-

[1] Note, however, that these results and observations do not prohibit the exixtence of an efficient solution.

[2] Data Base Management Systems and Federated Data Base Management System, respectively - Y.R.

[3] Typically, a RM is unaware of any resource state dependency with states of resources external to the RM, implied by applications (cross RM integrity constraints). This is also true in the cases where RMs are coordinataed by *multi-database systems*, which provide applications with integrated views of resources.

erable failures. It means that either a distributed transaction is *committed,* i.e., its effects on all the resources involved become permanent, or it is *aborted* (*rolled back),* i.e., its effects on all the resources are undone. The most commonly used atomic commitment protocols are variants of the *Two Phase Commitment* protocol (*2PC* - [Gray 78], [Lamp 76], [Moha 86]). Examples are *Logical Unit Type 6.2* of International Business Machines Corporation ([LU6.2]), *Digital Equipment Corporation's Distributed Transaction Manager - DECdtm* ([DECdtm]),and the *ISO - OSI* and *X/Open* standards for *Distributed Transaction Processing* ([OSI-DTP] and [X/Open-DTP]). A well known *local* (i.e., local to each RM) concurrency control mechanism that together with AC guarantees global serializability is *Strong Strict Two Phase Locking* (S-S2PL[1]; "release locks issued on behalf of a transaction only after the transaction has ended"). This fact has been known for several years, and has been the major correctness foundation for distributed transactions. Various technical documents about distributed transaction management (e.g., [OSI-CCR]) have mentioned it. The observation that local S-S2PL guarantees global serializability appears explicitly at least in [Pu 88], [Brei 90] and [Brei 91][2]. The disadvantage of this approach is that *all* the RMs involved have to implement S-S2PL based concurrency control, even if other types are preferable for some RMs. Another history property, *dynamic atomicity,* that generalizes S-S2PL and provides serializability, while allowing RM autonomy, is defined in [Weih 89][3]. Other properties defined there cannot be enforced globally by autonomous RMs, and require the RMs to have more knowledge about global concurrency control (e.g., transaction timestamps).

In this paper the relationships between histories of individual RMs and the global history that comprises them are examined, and the above properties are generalized. We define a history property named *Commitment Ordering (CO),* and show that guaranteeing it is a *necessary* and *sufficient* condition for *guaranteeing* global serializability under the conditions of RM autonomy. CO can be implemented as standalone serializability mechanisms as well as being incorporated with other concurrency control mechanisms. Since CO can be enforced solely by controlling the order of transactions' commit events, it can be combined with any other concurrency control mechanism without affecting the mechanism's resource access scheduling strategy. This means that any order of access operations by any serializable schedule is also allowed by some CO schedule (the CO constraint is on the commit partial order, and not on the access operations). Thus, CO allows selecting and optimizing concurrency control for each RM according to the nature of transactions involved. Enforcing CO does not require aborting more transactions than those needed to be aborted for global serializability violation prevention, which is determined exclusively by the resource access orders, and is independent of the commit orders. S-S2PL based RMs already provide CO, since S-S2PL is a special case of CO.

In summary, serializability of transaction histories across (any) different RM types, that may use different concurrency control mechanisms but provide the CO property, is guaranteed without any global coordination or services beyond AC. Thus, the CO solution is fully distributed and does not incure any communication overhead. In addition to providing an effective solution to the global serializability problem, CO based protocols are also expected to

[1] Since the S-S2PL mechanism defined here is generic, we use S-S2PL also as the implied history property's name; see also section 2.3.3.

[2] [Brei 91] uses the term *rigorousness* for S-S2PL.

[3] *Dynamic atomicity* ([Weih 89]) is interpreted in our transaction model as follows:
Let op_1 and op_2 be conflicting operations of two transactions T_1 and T_2, respectively, and c_1, c_2 their respective commit events. A history is *dynamic atomic* if for any two committed transactions T_1 and T_2, $op_1 < op_2$ implies that there exists an operation op_3 of T_2, distinct from c_2, such that $c_1 < op_3$. Dynamic atomicity is a special case of CO. No general algorithm for enforcing dynamic atomicity is given in [Weih 89].

provide performance advantage over existing S-S2PL based locking protocols, even in the single RM case (see also [Agra 91], [Agra 92], and [Raz 92a]).

Also [Brei 91] defines CO, naming it *strong recoverability* and uses it to show that applying S-S2PL locally provides global serializability. No algorithm for enforcing CO (beyond S-S2PL) is given there. [Brei 92] uses CO in a distributed environment in a way different from the way it is used in this work. Since in [Brei 92] the interpretation of the *commitment event* in the distributed case is different from the interpretation here, the conclusion of [Brei 92] is that applying CO locally does not imply global serializability, contrary to one of the main results of this work. [Brei 92] proposes a centralized CO based algorithm that requires a centralized *site-graph* and does not allow two or more concurrent global transactions at a same site.

Both S-S2PL and *commit-blocking* CO (CBCO, a special case of CO) are special cases of locking with *Ordered Sharing* (OS), defined in [Agra 90] and [Agra 91]. OS generalizes 2PL by modifying the blocking semantics of read and write locks in eight different variations (using eight locking schemes ("tables"), T_1,...,T_8, where T_1 corresponds to 2PL). Like 2PL, also OS is two phased, and uses the end of phase 1 as a serialization point (i.e., the order of any conflicting operations matches the order of these events in respective transactions). When all locks are released at transaction end, OS reduces to CBCO. [Agra 92] defines this special case of OS (i.e., the class CBCO) in terms of *commitment ordering.* (see more on OS in section 4.5 below.)

Section 2 is an overview and reformulation of serializability theory, which provides the foundation for analyzing CO. Section 3 defines CO and describes its properties. Section 4 examines CO schedulers and presents generic CO algorithms. Section 5 deals with multi RM histories, atomic commitment, relationships between local and global properties, global CO scheduling, and global deadlocks. Section 6 shows that CO is exactly the property required to guarantee global serializability across autonomous RMs. Section 7 provides a conclusion. The appendix describes how to use CO in existing distributed transaction processing architectures (based on [Raz 91a]). This paper is based on [Raz 90]. A short summary of major results appeared in [Raz 94]. Throughout the paper an attempt has been made to keep the presentation as intuitive and informal as possible, without sacrificing accuracy.

2 Histories and their properties - an overview

This section summarizes and reformulates known concepts and results of concurrency control theory (see also [Bern 87]), as well as introducing some new concepts, as a foundation for the following sections. The transaction model defined in this section maintains most of the semantics of the model defined in [Bern 87], and preserves its related terminology. Concepts are reformulated, to more conveniently express and prove the results derived later.

2.1 Transactions and histories

A *transaction*, informally, is an execution of a set of programs that access shared resources. It is required that a transaction is *atomic*, i.e., either the transaction completes successfully and its effects on the resources become permanent[1], or all its effects on the resources are undone. In the first case, the transaction is *committed.* In the second, the transaction is *aborted.* Formally, we use an abstraction that captures only events and relationships among them, that are necessary for reasoning about concurrency control:

A *single RM transaction* T_i is a *partial order* of events (specific events within the above informally defined transaction).
The (binary, asymmetric, transitive, irreflexive) relation which comprises the partial order is denoted "$<_i$" .

Remarks:

- $event_a <_i event_b$ reads: $event_a$ *precedes* $event_b$ (in T_i).
- The subscript i may be omitted when the transaction's identifier is known from the context.
- $<_i$ may be omitted for total orders (i.e., a transaction may be represented by an event sequence).

The events of interest are the following[2]:

- The *operation* of *reading* a resource; $r_i[x]$ denotes that transaction T_i has retrieved (read) the (partial) state[3] of the resource x.
- The *operation* of *writing* a resource; $w_i[x]$ means that transaction T_i has modified (written) the state of the resource x.
- *Ending* a transaction; e_i means that T_i has ended (has been either committed or aborted) and will not introduce any further operations.

A transaction obeys the following *transaction rules* (*axioms*):

- **TR1**
  A transaction T_i has exactly a single event e_i .
  A value is assigned to e_i : $e_i = c$ if the transaction is *committed*; $e_i = a$ if the transaction is *aborted*.
  Notation: e_i may be denoted c_i or a_i when $e_i = c$ or $e_i = a$, respectively.

[1] The term *permanent* is relative and depends on a resource's volatility (e.g., sensitivity to process or media failure).

[2] More event types such as locking and unlocking may be introduced when necessary.

[3] Each resource has one state at a time. We deal with resource states informally only.

- **TR2**

  For any operation $p_i[x]$ (either $r_i[x]$ or $w_i[x]$) $p_i[x] <_i e_i$

Two operations on a resource x, $p_i[x]$, $q_j[x]$ are *conflicting*, if they are *noncommutative*, i.e., applying them in different orders results in two different states[1] of x.
A more restrictive[2] approach assumes them to be conflicting, if at least one of them is a write operation.

A *complete history* H *over* a set T of transactions is a partial order with a relation $<_H$ defined according to the following *history rules* (*axioms*):

- **HIS1**

  If T_i is in T and $event_a <_i event_b$ then $event_a <_H event_b$

- **HIS2**

  If T_i and T_j are in T then for any two conflicting operations $p_i[x]$, $q_j[x]$,
  either $p_i[x] <_H q_j[x]$ or $q_j[x] <_H p_i[x]$

- **HIS3**

  Let T_i, T_j be transactions in T, where $e_i = a$.
  If $w_i[x] <_H r_j[x]$ then either $e_i <_H r_j[x]$ or $r_j[x] <_H e_i$
  (Without this rule a history's semantics (as reflected by resource states) is not uniquely determined, since if $e_i = a$ the effect of $w_i[x]$ is undone; i.e., reading x after e_i results in retrieving the last state of x that was written by other (unaborted) transaction than T_i.)

Remarks:

- The subscript H in $<_H$ may be omitted when H is known from the context.
- The graphic symbol $\rightarrow_H$ may be used instead of $<_H$ when convenient.
- $<_H$ may be omitted for total orders, i.e., a history may be represented by an event sequence.

For modeling executions with incomplete transactions, we define a *history* to be any *prefix*[3] of a complete history.

2.2 On guaranteeing a property

In the following sections we examine conditions for *guaranteeing* that a system (any collection of interacting components or objects) generates histories with certain properties. This concept is formalized as follows:

[1] Distinguishing states, and thus, operation commutativity, may depend on the RM's semantics.

[2] Two write operations on the same resource may commute, e.g., *increment* and *decrement* of a *counter*.

[3] A *prefix* of a partial order P over a set S is a partial order P' over a set $S' \subseteq S$, with the following properties:
If $b \in S'$ and $a <_P b$ then also $a \in S'$
If $a,b \in S'$ then $a <_P b$ if and only if $a <_P, b$

Definition 2.1

Let S_A be the set of all reachable states of a system A.

The system A *guarantees* a property P, if every state in S_A has property P.

§

It is often claimed later that guaranteeing a property P_1 is *necessary (a necessary condition) to guarantee* property P_2. This means that, if *all* the reachable states have property P_2 (are in P_2), then they also have property P_1. Equivalently, the existance of a reachable state that does not have property P_1 implies the existance of a (possibly different) reachable state that does not have property P_2.

We concentrate on the case where systems' states are histories generated by the system[1].

2.3 History classes

Remark: A property's acronym is also used as the name for the class of all histories with this property.

2.3.1 Serializability

Let T_1 and T_2 be two distinct transactions. Transaction T_2 *is in a conflict with* transaction T_1, if $p_1[x] < q_2[x]$ for respective conflicting operations $q_2[x]$, $p_1[x]$.

The conflict types are *ww*, *wr*, *rw*, when $p_1[x]$, $q_2[x]$ are write-write, write-read, and read-write operations, respectively.

Remark: Note the asymmetry in the definition above.

There is a *conflict equivalence* between two histories H and H' (the two are *conflict equivalent*) if they are both defined over the same set of transactions T, and consist of the same transaction events (for partially executed transactions), and

$p_i[x] <_H q_j[x]$ if and only if $p_i[x] <_H, q_j[x]$

for any *conflicting* operations $p_i[x]$, $q_j[x]$ of any *committed* transactions T_i, T_j, respectively, in T (i.e., H and H' have the same conflicts between operations of committed transactions).

A history H over a transaction set T is *serial*, if for every two transactions T_i, T_j in T all the operations and the end of T_i precede all the operations and the end of T_j (i.e., if $p_i[x] <_H q_j[y]$ then for any operations $s_i[u]$, $t_j[v]$ in H, $s_i[u] <_H t_j[v]$, and $e_i <_H t_j[v]$).

The *commit projection* of a history H, is its *projection* (*restriction*)[2] on its set of committed transactions.

A history is *serializable* (SER; is in SER), if its commit projection is conflict equivalent to some serial history.

[1] The related state transition function has a history and an event set as arguments. Its values on a given history and its prefixes, and on a given event set and its subsets, are compatible. Such a function is neither formalized nor explicitly used in this work.

[2] Let P be a partial order over a set S. A *projection (restriction)* of P on a set $S' \subseteq S$ is a partial order P', a subset of P, that consists of all the elements in P, involving elements of S' only.

Transaction states (in addition to *committed* and *aborted*) are defined as follows:

A transaction is *decided*, if it is either *aborted* or *committed;* otherwise, it is *undecided.*

An undecided transaction is *ready* if it has completed its processing, and is prepared either to be committed or aborted; otherwise it is *active.*

The following diagram defines the possible transitions between states:

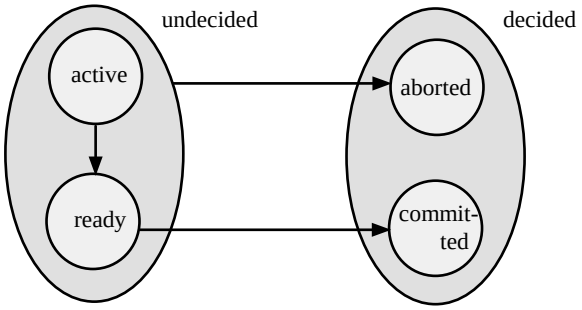


Figure 2.1: Transaction states and their transitions

The *Serializability Graph* of a history H, SG(H), is the following directed graph:

SG(H) = (*T*, *C*) where

- *T* is the set of all unaborted (i.e., committed and undecided) transactions in H
- *C* (a subset of *T*x*T*) is a set of edges that represent transaction conflicts:
  Let T_1, T_2 be any two transactions in *T*.
  There is an edge from T_1 to T_2 if T_2 is in a conflict with T_1.

The *Committed Transaction Serializability Graph* of a history H, CSG(H), is the subgraph of SG(H) with all the committed transactions as nodes and all the respective edges.

The *Undecided Transaction Serializability Graph* of a history H, USG(H), is the subgraph of SG(H) with all the undecided transactions as nodes and all the respective edges.

Theorem 2.1 provides a criterion for checking serializability:

Theorem 2.1 - The Serializability Theorem

A history H is *serializable* (in SER) if and only if CSG(H) is cycle-free.

(For a proof see, for example, [Bern 87]).

2.3.2 Recoverability

This section defines history properties that guarantee certain desired behavior patterns when aborts occur (see also [Bern 87], [Hadz 88]).

Recoverability is an essential property of histories when aborted transactions are present (i.e., in all real situations). Recoverability guarantees that *committed* transactions read only resource states written by committed transactions, and hence, no committed transactions read corrupted states. Recoverability also ensures that a serializable history has the same semantics (i.e., the history's outcome as reflected by the resources' states) as a conflict-equivalent serial history (when exists; e.g., for complete histories). This may not be true without recoverability, if aborted transactions are present. The following definition of recoverability is equivalent to the one in [Hadz 88], and to the informal description of recoverability in [Bern 87] (the formal definition in [Bern 87] does not provide a correctness criterion for schedules with aborted transactions).

Let T_1 and T_2 be two distinct transactions. We say that a transaction T_2 *reads* (a resource x) *from* (in a *read-from* conflict, or *wrf* conflict with; *wrf* is a special case of a *wr* conflict) transaction T_1 if T_2 reads x before T_1 is aborted (if aborted), and T_1 is the last transaction to write x before being read by T_2 (i.e., $w_1[x] < r_2[x]$ and there is no event t such that $w_1[x] < t < r_2[x]$, where t is either a_1 or $w_3[x]$ of some T_3).

It is required that for any two transactions T_1, T_2 in H, whenever T_2 reads any resource from T_1, aborting T_1 implies aborting T_2 (i.e., (T_2 reads from T_1) implies (e_1 = a implies e_2 = a)).
To guarantee this, T_2 should be decided only after T_1 has been decided (this is a necessary condition[1]). Thus, a history H is defined to be *recoverable* (REC; in REC) if for any two transactions T_1, T_2 in H, whenever T_2 reads any resource from T_1, T_1 ends before T_2 does ($e_1 < e_2$), and aborting T_1 implies aborting T_2.
Formally, (T_2 reads from T_1) implies ($e_1 < e_2$ and (e_1 = a implies e_2 = a)).
The above formulation of recoverability allows it to be enforced effectively.

Aborts caused by transactions reading states written by aborted transactions (*cascading aborts*) are prevented if any transaction in H reads only data written by already committed transactions (i.e.,
(T_2 reads from T_1) implies e_1 = c). *Avoiding cascading aborts* (ACA; *cascadelessness*) is the property which is necessary and sufficient to guarantee the above condition[2]: H *is ACA* (in ACA), if for any two transactions T_1, T_2 in H, (T_2 reads x from T_1) implies (e_1 = c and $e_1 < r_2[x]$).

Let T_1, T_2 be any two transactions in H. H is *strict* (ST; is in ST; has the *strictness* property) if

$w_1[x] < p_2[x]$ implies $e_1 < p_2[x]$, where $p_2[x]$ is either $r_2[x]$ or $w_2[x]$.

Strictness simplifies the restoration of a resource's state after aborting transactions that have written that resource. The recovery procedures of most existing database systems rely on strictness.

[1] The claim is proven by assuming the contrary, i.e., $e_2 < e_1$, and having T_2 committed, while T_1 is later aborted.

[2] Sufficiency is obvious. Necessity is proven by assuming $r_2[x] < e_1$ and having T_1 aborted.

Theorem 2.2 follows immediately from the definitions above:

Theorem 2.2 ([Bern 87], [Hadz 88])

REC $\supset$ ACA $\supset$ ST where "$\supset$" denotes a strict containment.

2.3.3 Two Phase Locking

Two Phase Locking (2PL) is a serializability mechanism that implements two types of locks[1]: *write locks* and *read locks.* A write lock on a resource blocks both read and write operations of that resource, while a read lock blocks write operations only. 2PL consists of partitioning a transaction's duration to two *phases*: in the first, locks are acquired; in the second, locks are released ([Eswa 76]).

A history is defined to be a *2PL history* (it is in the class 2PL), if it can be generated by the 2PL mechanism.

Combining strictness (ST) with 2PL results in *Strict Two Phase Locking* (S2PL = ST$\cap$2PL). To enforce S2PL, *write locks* issued on behalf of a transaction are not released until its end. *Read lock*s, however, can be released earlier, after the end of phase one of 2PL.

The property *Strong-S2PL*[2] (S-S2PL) requires that all locks are not released before the transaction ends (either committed or aborted).

Formally: A history H is S-S2PL (in S-S2PL), if for any conflicting operations $p_1[x]$, $q_2[x]$ of distinct transactions T_1, T_2, respectively, in H, $p_1[x] < q_2[x]$ implies $e_1 < q_2[x]$.

Theorem 2.3 summarizes the relationships among the 2PL classes (follow by the definitions):

Theorem 2.3

2PL $\supset$ S2PL $\supset$ S-S2PL

2.4 On inherently-blocking and noninherently-blocking properties

Some history properties can be enforced only by blocking mechanisms. A mechanism is *blocking*, if in some situations it delays some transaction's *event* until a certain *event*(s) occurs in some *other* transaction(s).

A mechanism is *operation-blocking*, if in some situations it delays a transaction's *operation* until a certain *event*(s) occurs in some *other* transaction(s), or aborts all transactions with blocked operations (to avoid operation-blocking, that otherwise would occur).

We define a history property to be *inherently-blocking*, if it can be enforced by operation-blocking mechanisms only. Otherwise it is *noninherently-blocking*.

Both serializability and recoverability are noninherently-blocking, since they can always be guaranteed by aborting a violating transaction any time before it ends, without having any operations blocked. This observation is the basis for *optimistic concurrency control* ([Kung 81]), where transactions run without blocking each other's operations,

[1] A lock is considered any mechanism that blocks resource-access operations.

[2] This property is characterized in [Bern 87] as a special case of Strict-2PL, but is not given any distinct name.

and are aborted, if they violate serializability or any other desired property. 2PL, ACA and their special cases, on the other hand, are inherently-blocking.

Note that the mutual blocking of two or more transactions is the cause of *deadlock* situations. Thus, nonblocking mechanisms guarantee *deadlock-freeness.*

Remark: In this work we deal with blocking and deadlocks informally only.

2.5 On commit-decision delegation

In some situations the decision whether to commit or abort a ready transaction is delegated from one system (object, component) to another system (object, component) via a notification. This notification is denoted as a *YES vote on the transaction*.

Definition 2.2

Let transaction T be in the *ready* state. System A *delegates* the commit decision on a transaction T to system B by *voting* YES *on* T, if system A is prepared to either commit T or abort it, according to the decision taken by system B. After voting YES system A cannot affect the decision anymore.

Remark: System A can abort transactions. It cannot vote YES on a transaction after aborting it.

Let y_i denote the YES voting event by system A on transaction T_i, and d_i the decision event that takes place in system B. d_i takes the values a or c, and may be denoted a_i or c_i respectively (when a distinction between e_i and d_i is clear by the context). The following *commit decision delegation (CDD) rules (axioms)* involving these events hold true:

- **CDD1**

  $p_i[x] < y_i$ for any operation $p_i[x]$ of T_i (i.e., all the transaction's operations are completed before voting YES on the transaction[1]).

- **CDD2**

  $y_i < d_i$ (i.e., when commit-decision delegation is applied, an explicit vote is required *before* any decision to commit or abort can be made).

- **CDD3**

  $d_i < e_i$ (i.e., the transaction is ended by system A only after being notified of the decision).

- **CDD4**

  $e_i = c$ if and only if $d_i = c$ (the obedience rule).

- **CDD5**

  event $< d_i$ implies event $< y_i$ for any event $\neq y_i$ in system A (i.e., all such precedence dependencies with d_i are through y_i).

[1] Commiting T_i by system A *after* the decision is made may involve the completion of *write* operations that have been written before the voting to a temporary storage, and not to the resource itself. However, in such cases the resource is locked for *any* operation until the transaction ends, and thus CDD1 can be assumed also for this case.

- **CDD6**

  $d_i <$ event implies $e_i <$ event for any event $\neq e_i$ in system A (i.e., all such precedence dependencies with d_i are through e_i).

§

Note that CDD1,2,3 are consistent with TR2. CDD5,6 mean that the interactions between systems A and B are reflected in the generated histories through the voting mechanism only.

In some situations, where dependencies exist between decision events of different transactions (see sections 4.4 and 5 below), the following condition is needed to guarantee such dependencies:

Definition 2.3

A system (object) that *delegates* commit decisions *obeys* the *commit-decision delegation, dependency condition* (*CD*^{3}*C) for transactions* T_1 and T_2 , if it votes YES on T_2 only after commiting or aborting T_1 , i.e., the following relationship holds true:

- **CD^{3}C**

  $e_1 < y_2$

§

Note the asymmetry in the definition above.

Theorem 2.3 summarizes the conditions for decision event dependencies. It reflects that the delegating system, system A,when voting, does not have the knowledge about the decision to be taken by system B.

Theorem 2.3

Let system A delegate the commit decision on transactions T_1 and T_2 to system B.

Then CD^{3}C for T_1 and T_2 (i.e. $e_1 < y_2$) is a necessary and sufficient condition for $d_1 < d_2$.

Proof:

(i) $d_1 < d_2$ implies $d_1 < y_2$ by CDD5. However $d_1 < y_2$ implies $e_1 < y_2$ (i.e., CD^{3}C) by CDD6.

(ii) $d_1 < e_1$ (CDD3) and $y_2 < d_2$ (CDD2), and thus $d_1 < d_2$ follows by CD^{3}C.

§

3 Commitment Ordering (CO)

3.1 CO as a serializability concept

Commitment Ordering (CO) is a property of histories that guarantees serializability. It generalizes S-S2PL. A history is *CO* if the order (<) of any two conflicting operations in any two *committed* transactions matches the order of the respective commit events.

After a transaction accesses a resource, S-S2PL blocks any conflicting operations on the resource until the end of that transaction. CO, on the other hand, allows access by conflicting operations, while using any access scheduling strategy. This allows CO to be implemented also in a nonblocking manner, which guarantees *deadlock-freeness.* The price for this, however, is the possibility of *cascading aborts* when recoverability is applied.

Definition 3.1

A history is in CO if for any conflicting operations $p_1[x]$, $q_2[x]$ of any *committed* transactions T_1, T_2 respectively , T_2 being in a conflict with T_2 implies $e_1 < e_2$.

Formally:

$(e_1 = c \text{ and } e_2 = c \text{ and } (T_2 \text{ is in a conflict with } T_1))$ implies $e_1 < e_2$.

§

The following property is a useful special case of CO:

Definition 3.2

A history is in *commit-blocking* CO (CBCO), if it is in CO and for *every* two transactions T_1, T_2 where T_2 is in a conflict with T_1, T_2 is committed only after T_1 is decided.

§

We now show that CO implies serializability:

Theorem 3.1

SER $\supset$ CO (strict containment).

Proof:

(i) Let a history H be in CO, and let $... \rightarrow T_i \rightarrow ... \rightarrow T_j \rightarrow ...$ be a (directed) path in CSG(H). By the CO definition (3.1) and an induction by the order on the path above, we conclude that $c_i < c_j$.

(ii) Now, suppose that H is not in SER.

By theorem 2.1 (without loss of generality) there is a cycle
$T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ in CSG(H), where $n \geq 2$.

First, let T_i and T_j in (i) be T_1 and T_2 above, respectively (consider an appropriate prefix of the expression representing the cycle above).
This implies by (i) that $c_1 < c_2$.

Now, let T_i and T_j in (i) be T_2 and T_1 above, respectively (consider an appropriate suffix of the expression representing the cycle above). This implies that $c_2 < c_1$.

However, $c_1 < c_2$ and $c_2 < c_1$ contradict each other, since the relation "<" is asymmetric.

Hence CSG(H) is acyclic, and H is in SER by Theorem 2.1.

Now examine the following serializable, non CO history to conclude that the containment is strict:

$r_1[x]\ w_2[x]\ c_2\ c_1$

§

Though CO is a proper subset of SER, these two classes do not differ with respect to resource access (read and write) operations. I.e., if the commit and abort events in the histories are removed, CO and SER induce identical sets of modified histories. This fact is implied by lemma 3.1:

Lemma 3.1 - The CO Commit Delay Lemma

For every history H in SER a history H' exists in CO with the following properties.

- Both H and H' are defined over the same set of transactions and operations.
- The last state (e.g., committed, aborted, etc.) of every transaction in H is identical to that in H'.
- For every two conflicting operations $p_1[x], q_2[x]$, $p_1[x] <_H q_2[x]$ if and only if $p_1[x] <_{H'} q_2[x]$.

Proof:

Start with H'=H. "Delay" commit events in H' when required to comply with CO, i.e., change the respective "<" relation elements as follows:

Let H" be a serial history that is conflict-equivalent to the commit projection of H (the existence of H" is guaranteed by the definition of serializability, and H being in SER; see section 2). Apply the commit (total) order of H" to H' inductively. Without loss of generality let c_n be the n-th commit event in H". Reppeat the following step for c_{n+1} starting from n=1: If $c_n < c_{n+1}$ is not true in H', then "delay" c_{n+1} by replacing all the < elements in H' involving c_{n+1} in a way that $c_n < c_{n+1}$, and H' remains a partial order (this is possible since initially H' is a partial order). Note that by delaying c_{n+1} rule T2 (in section 2) is maintained, and the resulting H' is a history. After completing the process for all the commit events, the resulting H' is in CO. Since no $op_1 < op_2$ elements in H', where op_1 and op_2 are read or write operations, have been changed, the lemma follows.

§

We use the notation in definition 3.3 to formalize the above observation:

Definition 3.3

Let H be a history. Then H_{op} is the restriction of H on read and write operations.

Let X be a class of histories. Then X_{op} is the class generated from X by replacing its histories with their restrictions on read and write operations (i.e., $X_{op} = \{H_{op} \mid H \text{ is in } X\}$).

§

Theorem 3.2 follows by theorem 3.1 and lemma 3.1:

Theorem 3.2

$CO_{op} = SER_{op}$

Theorem 3.2 means that all the operation (partial) orders allowed in SER histories are also allowed in CO histories, and vise versa.

3.2 CO - recoverable histories

Since recoverability is a correctnes criterion when aborted transactions are present, the class CO∩REC is of a special interest. The fact that both CO and recoverability are noninherently-blocking, suggests that histories in this class can be generated by nonblocking mechanisms, which provide *deadlock-freedom*. Section 4 below describes such a generic mechanism.

First, note that recoverability is incomparable with CO (i.e., one does not imply the other):

Theorem 3.3

REC and CO are incomparable

Proof:

The history $w_1[x]\ r_2[x]\ a_1\ c_2$ is in CO and not in REC;
The history $r_1[x]\ w_2[x]\ c_2\ c_1$ is in REC and not in CO.

§

The following corollary is a consequence of theorem 3.3:

Corollary 3.1

- CO ⊃ CO∩REC
- REC ⊃ CO∩REC

The ACA (cascadelessness) property blocks operations that create write-read conflicts until the end of the preceding transaction involved with the conflict. This means that every mechanism that generates histories both in CO and ACA is blocking. This fact and the following results mean that the CO∩ACA property and all its special cases down to S-S2PL (see section 3.3 below) generate a hierarchy of inherently-blocking properties. These properties impose increasing number of constraints and blocking conditions as we move down the hierarchy.

The following relationships exist as well:

Theorem 3.4

1. ACA ⊃ CO∩ACA
2. CO∩REC ⊃ CO∩ACA
3. ST ⊃ CO∩ST

4. CO∩ACA ⊃ CO∩ST

5. CO∩ST ⊃ CO∩ST∩2PL

6. S2PL (= ST∩2PL) ⊃ CO∩ST∩2PL

Proof:

The containment relationships are implied by the definition of intersection and by the containment relationships among REC, ACA and ST (theorem 2.2). The containment relationships are strict, as implied by the following respective examples:

1. The history $r_1[x]\ w_2[x]\ c_2\ c_1$ is in ACA and not in CO∩ACA.
2. The history $w_1[x]\ r_2[x]\ a_1\ a_2$ is in CO∩REC and not in CO∩ACA.
3. The history $r_1[x]\ w_2[x]\ c_2\ c_1$ is in ST and not in CO∩ST.
4. The history $w_1[x]\ w_2[x]\ a_1\ a_2$ is in CO∩ACA and not in CO∩ST.
5. The history $r_1[x]\ w_2[x]\ r_2[y]\ r_1[y]\ a_1\ a_2$ is in CO∩ST and not in CO∩ST∩2PL.
6. The history $r_1[x]\ w_2[x]\ c_2\ c_1$ is in ST∩2PL and not in CO∩ST∩2PL.

§

Finally we show that S-S2PL implies CO (by proving a stronger result):

Theorem 3.5

CO∩ST∩2PL ⊃ S-S2PL

Proof:

If a history H is in S-S2PL, it is in S2PL=ST∩2PL (theorem 2.3), and hence in ST and 2PL. Being in S-S2PL all locks on behalf of a transaction T_1 in H are being released only after the transaction ends, and, thus, no conflicting operations of any T_2 can be issued before T_1 ends. Hence, transaction T_2 cannot end before T_1 does, and the history is also in CO. Formally, H being in S-S2PL means that $p_1[x] < q_2[x]$, where $p_1[x]$, $q_2[x]$ are conflicting operations of T_1 , T_2 , respectively, imply $e_1 < q_2[x]$. Hence, $q_2[x] < e_2$ (axiom TR2) implies $e_1 < e_2$. This is especially true when e_1 = c and e_2 = c. Thus H is in CO.

Since $r_1[x]\ w_2[x]\ a_1\ a_2$ is in CO∩ST∩2PL but not in S-S2PL, the containment is strict.

§

3.3 Relationships among properties - a summary

The following diagram summarizes the containment relationships between history classes, as proven above.

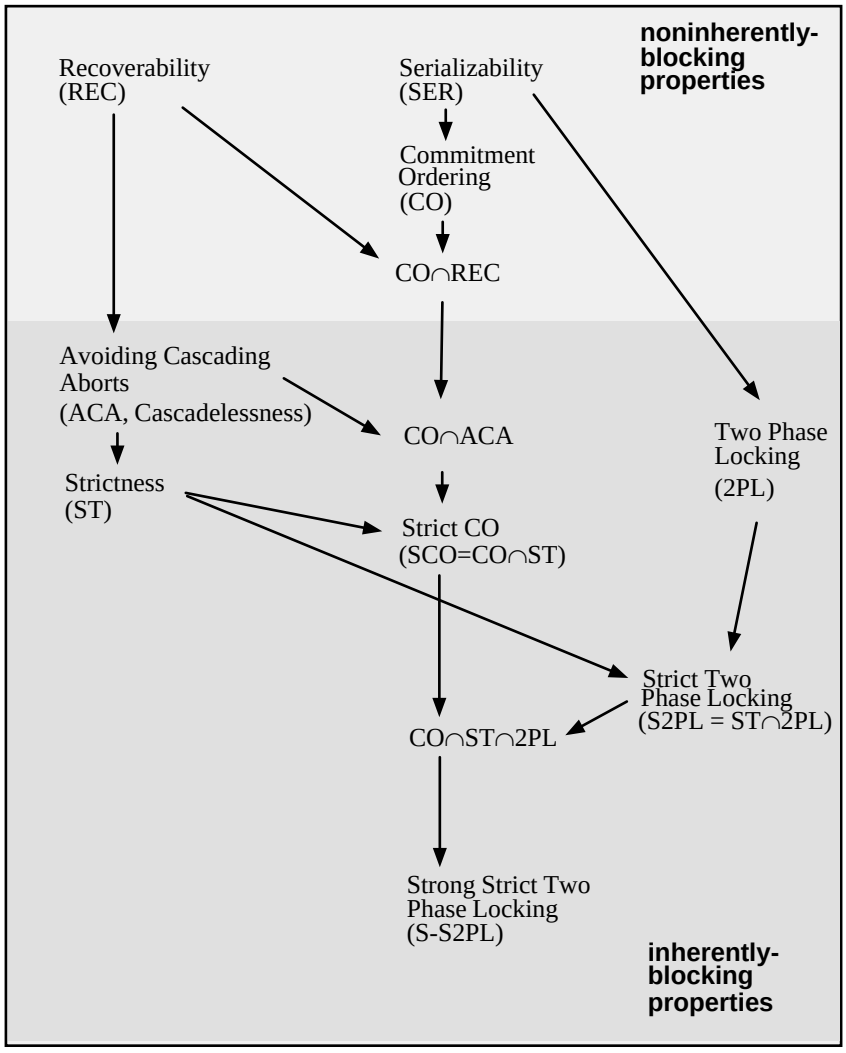


Figure 3.1: Class containment relationships

An arrow from a class A to a class B indicates that class A strictly contains B; a lack of a directed path between classes means that the classes are incomparable.

A property is *inherently-blocking* if it can be enforced only by blocking transaction's operations until certain events occur in *other* transactions.

4 Commitment Ordering schedulers

A *scheduler* is a RM's component that schedules certain transactions' events. *Commitment-Ordering (CO) schedulers* are schedulers that guarantee generating CO histories. Generic mechanisms, that can be combined in various ways to implement CO schedulers, are presented in this section. The algorithms described below provide algorithmic characterizations for the properties CO, CO∩REC, and CO∩ST (SCO). The algorithms' graph-theoretic description is general. Real-life implementations, particularly of shcedulers forproperty subsets may take a different shape (e.g., locking).

4.1 Schedulers: components and classification

Schedulers typically deal with three types of transaction events:

- Transaction initiation
- Resource access
- Transaction termination

Concentrating on the latter two types[1], we model a (complete) scheduler as consisting of two components:

- Resource Access Scheduler (RAS)
  A component that manages the resource access requests arriving on behalf of transactions, and decides when to execute which resource access operation.
- Transaction Termination Scheduler (TTS)
  A competent that monitors the set of transactions and decides when and which transaction to commit or abort. In a multi RM environment this component participates in *atomic commitment* procedures on behalf of its RM and controls (within the respective RM) the execution of the decision reached via atomic commitment for each relevant transaction.

A scheduler component is *blocking* if it executes certain transaction's event requests only after certain events have occurred in some *other* transaction(s). Otherwise, it is *nonblocking.*

Nonblocking schedulers implement the so called *optimistic concurrency control* approach ([Kung 81]). When a scheduler is *nonblocking*, it provides *deadlock-free* executions.

4.2 A "pure" CO TTS - The Commitment Order Coordinator (COCO)

The following TTS type, the *Commitment Order Coordinator (COCO),* checks for CO only and generates CO histories. The generated histories are not necessarily recoverable. Recoverability, if required, can be applied by enhancing the COCO (see section 4.2) or by an external mechanism (e.g., see section 4.4).

[1] A transaction is usually initiated as soon as computing resources are available, without considering the effect on the history's properties. Situations where an advantage may be taken of controlling initiation are not dealt with here.

A COCO maintains a serializability graph, the USG, of all *undecided* transactions. Every new transaction processed by the RM is reflected as a new node in the USG; every conflict between transactions in the USG is reflected by a directed edge (an edge between two transactions may represent several conflicts).

USG(H) = *(UT,C)* where

- *UT* is the set of all undecided transactions in a history H
- *C* (a subset of *UT*x*UT*) is the set of directed edges between transactions in *UT*. There is an edge from T_1 to T_2 , if T_2 is in a conflict with T_1.

The set of transactions in the USG, aborted as a result of committing a transactionT (to prevent any future commitment ordering violation) is defined as follows:

$ABORT_{CO}(T) = \{ T' \mid$ The edge $T' \rightarrow T$ is in $C \}$

These aborts cannot be compromised, as stated by lemma 4.1:

Lemma 4.1

Aborting all the members of $ABORT_{CO}(T)$, after T is committed, is necessary for guaranteeing CO (assuming that every transaction is eventually decided).

Proof:
Suppose that T is committed. Let T' be some transaction in $ABORT_{CO}(T)$. Thus T' is undecided when T is committed. If T' is later committed, then c < c', where c and c' are the commit events of T and T' respectively. However, T is in a conflict with T', and thus, CO is violated.

§

Lemma 4.1 is the key for the CO algorithm.

Algorithm 4.1 - The CO Algorithm

Repeat the following steps:

- Select any transaction in the *ready* state (i.e., a transaction that has completed processing) T in the USG (using any criteria, such as by priorities assigned to each transaction; a priority can be changed dynamically as long as the transaction is in the USG), and commit it.
- Abort all the transactions in the set $ABORT_{CO}(T)$, i.e., all the transactions (both *ready* and *active*) in the USG that have an edge going to T .
- Remove any *decided* transaction (T and the aborted transactions) from the graph (they do not belong in the USG by definition).

Remark: During each iteration the USG should reflect *all* operations' conflicts until commit.

§

Example 4.1

The following figure demonstrates one iteration of the algorithm:

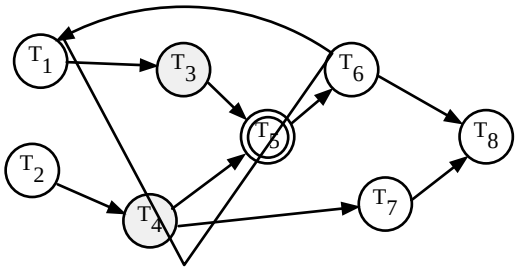


Figure 4.1: A USG

If T_5 is selected to be committed, then T_3 and T_4 are aborted;
T_3, T_4 and T_5 are then removed from the graph.

§

The following theorem states the algorithm's correctness:

Theorem 4.1

Histories generated by a scheduler involving a COCO (algorithm 4.1) are in CO.

Proof:

The proof is by induction on the number of iterations by the algorithm, starting from an empty history H_0 , and an empty graph $USG_0 = USG(H_0)$. H_0 is CO.

Let H_n be the history reflecting all the events before commiting the n-th transaction (n>0) by the algorithm. Then also H_1 is trivially in CO. $USG_n = USG(H_n)$ (its *UT* component) includes all the undecided transactions in H_n.

Assume that the history H_n is in CO. Now perform an additional iteration, number n+1, and commit transaction T_1 (without loss of generality - wlg) in USG_n. H_{n+1} includes all the transactions in H_n and the new (undecided) transactions that have been generated after completing step n (and are in USG_{n+1}).

Examine the following cases after completing iteration n+1:

- Let T_2, T_3 (wlg) be two committed transactions in H_n. If T_3 is in a conflict with T_2 then $c_2 < c_3$ since H_n is CO by the induction hypothesis.
- Obviously, $c_2 < c_1$ for every (previously) committed transaction T_2 in H_n with which T_1 is in a conflict.
- Suppose that a committed transaction T_2 is in a conflict with T_1. This means that T_1 is in $ABORT_{CO}(T_2)$, and thus aborted when T_2 was committed. A contradiction.

The cases above exhaust all possible pairs of conflicting committed transactions in H_{n+1}. Hence, H_{n+1} is CO.

§

By lemma 4.1 and theorem 4.1 we conclude that algorithm 4.1 characterizes the entire CO class (i.e., can generate all the CO histories). By theorems 3.1 and 4.1 we conclude the following:

Corollary 4.1

Histories generated by a system that includes a COCO are serializable.

Note that aborting the transactions in $ABORT_{CO}(T)$ when committing T prevents any cycle involving T being generated in the CSG in the future. This observation is a direct way to show that the algorithm 4.1 guarantees serializability. If a transaction exists, that does not reside on any cycle in the USG, then a transaction T exists with no incoming edges *from* any other transaction. T can be committed without aborting any other transaction since $ABORT_{CO}(T)$ is empty. If all the transactions in the USG are on cycles, at least one transaction has to be aborted when committing another one. If the COCO is combined with a scheduler (see also section 4.4 below) that guarantees (local) serializability, cycles in the USG are either prevented, or eliminated by the scheduler aborting transactions.

The histories generated by algorithm 4.1 are in the commit-blocking CO (CBCO) class if a transaction T is committed only when $ABORT_{CO}(T)$ is empty.

In a multi-RM environment, a COCO decides when to vote YES on a transaction in an atomic commitment (AC) protocol, rather than deciding when to commit it. After a notification to commit has arrived, when committing a transaction, the actions taken by the COCO are the same as for a single RM. When using AC, delaying the voting or blocking it usually reduces the number of aborted transactions (see more details in section 5 below).

4.3 The CO-Recoverability Coordinator (CORCO)

A CORCO is a CO TTS generating histories that are both CO and recoverable. This TTS is an enhancement of the COCO (section 4.2 above), and differs from it only in processing additional information to guarantee recoverability, and thus, possibly aborting additional transactions, to prevent recoverability violations.

A CORCO maintains an enhanced serializability graph, *wrf*-USG:

wrf-USG(H) = $(UT, C \cup C_{wrf})$ where

- *UT* is the set of all undecided transactions in the history H.
- *C* is a set of edges between transactions in *UT*:
  There is a *C* edge from T_1 to T_2 , if T_2 is in a conflict (conflicts) with T_1
  but has not *read from* T_1 (i.e., in any conflict other than *wrf*; see section 2.3.2).
- C_{wrf} is a set of edges between transactions in *UT* as well:
  There is a C_{wrf} edge from T_1 to T_2 , if T_2 has *read from* (in a *wrf* conflict with) T_1
  (and possibly is also in conflicts of other types with T_1).

Note that *C* and C_{wrf} are disjoint.

The set of transactions aborted as a result of committing T (to prevent future CO violation) is defined as follows:

$ABORT_{CO}(T) = \{ T' \mid T' \rightarrow T \text{ is in } C \text{ or } C_{wrf} \}$

The above definition of $ABORT_{CO}(T)$ has the same semantics as the definition of $ABORT_{CO}(T)$ for the COCO.

The set of aborted transactions due to recoverability, as a result of aborting transaction T', is defined as follows:

$$ABORT_{REC}(T') = \{ T'' \mid T' \rightarrow T'' \text{ is in } C_{wrf} \text{ or } T''' \rightarrow T'' \text{ is in } C_{wrf} \text{ where } T''' \text{ is in } ABORT_{REC}(T') \}$$

Note that the definition is recursive. This reflects the nature of cascading aborts.

A CORCO executes the following algorithm:

Algorithm 4.2 - The CO-REC Algorithm

Repeat the following steps:

- Select any *ready* transaction T in the *wrf*-USG, that does not have any incoming C_{wrf} edge (i.e., such that T is not in $ABORT_{REC}(T')$ for any transaction T' in $ABORT_{CO}(T)$; this avoids the need to later abort T itself), and commit it.
- Abort all the transactions T' (both *ready* and *active)* in $ABORT_{CO}(T)$.
- Abort all the transactions T'' (both *ready* and *active)* in $ABORT_{REC}(T')$ for every T' aborted in the previous step (cascading aborts).
- Remove any *decided* transaction (T and all the aborted transactions) from the graph.

Remarks:

- During each iteration the *wrf*-USG should reflect *all* operations' conflicts till commit.
- An *isolated wrf cycle* is a *wrf*-USG cycle, that consist of C_{wrf} edges only, and no transaction on it has an outgoing *C* edge *to* any transaction not on the cycle,. All transactions on isolated cycles are aborted asynchronously (the steps above neither commit nor abort such transactions; CO and serializability are violated if all the transactions are committed; recoverability is violated if not all of them either committed or aborted).

§

Example 4.2

The following figure demonstrates one iteration of the algorithm:

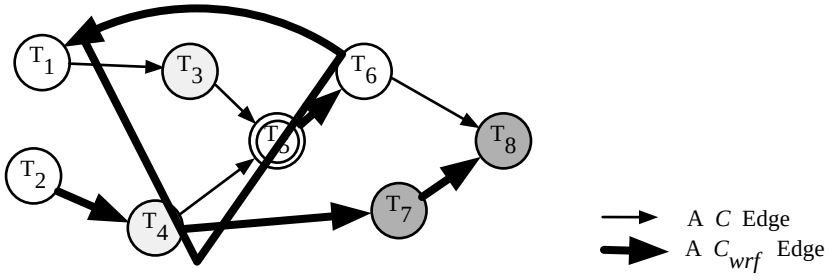


Figure 4.3: A *wrf*-USG

If T_5 is selected to be committed then T_3, T_4, T_7, T_8 are aborted;
all committed and aborted transactions are then removed from the graph.

$ABORT_{CO}(T_5) = \{ T_3, T_4 \}$
$ABORT_{REC}(T_3) = \phi$ (empty set)
$ABORT_{REC}(T_4) = \{ T_7, T_8 \}$

§

The algorithm's correctness is stated as follows:

Theorem 4.2

Histories generated by a scheduler involving a CORCO (algorithm 4.2) are CO and recoverable.

Proof:

The histories generated are CO by theorem 4.1, since a CORCO differs from a COCO only in possibly aborting additional transactions during each iteration (due to the recoverability requirement).

Since all the transactions that can violate recoverability (transactions in $ABORT_{REC}(T')$ for every aborted transaction T' in $ABORT_{CO}(T)$) are aborted during each iteration (i.e., transactions that read resources written by an aborted transaction before the abort), the generated histories are recoverable.

§

By theorems 3.1 and 4.2 we conclude the following:

Corollary 4.2

Histories generated by a system that includes a CORCO are both serializable and recoverable.

4.4 Combining a CO TTS with a RM's scheduler

The CO TTSs above can be combined with any RAS or a complete scheduler. When a CO TTS is combined with a scheduler, the scheduler delegates the commit decision (see section 2.5 above) to the CO TTS. If all the components are *non-blocking*, then also the combined mechanism is non-blocking, and hence ensures *deadlock-freeness*:

Corollary 4.3

There exist schedulers incorporating a COCO or CORCO that generate deadlock free executions only.

By Corollaries 4.1 and 4.2 the combined RAS (or scheduler) does not necessarily need to produce serializable histories in order to guarantee serializability, since the CO TTSs above take care of this.

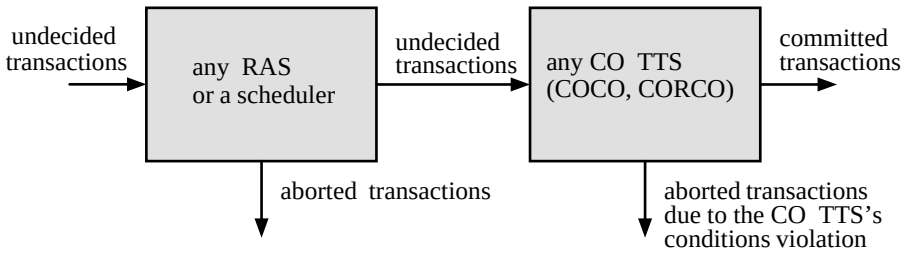


Figure 4.3

The commit/abort decision process using any CO TTS
combined with a RAS or scheduler

The combined mechanism executes as follows (See Figure 4.3 above):

First, a transaction interacts with a RAS (or a scheduler). Then if unaborted, when ready, the transaction is considered by a CO TTS as a candidate to be committed. A transaction may be aborted to prevent CO TTS's condition violation.

An important property of the CO TTSs is their total passivity with regard to resource access operations. The RAS (or the complete scheduler) can implement any resource access scheduling strategy without being affected by a combined CO TTS. The only requirement is that a CO TTS has updated conflicts information (e.g., a USG or a *wrf*-USG).

CO can be enforced on a given operation scheduling by the CO TTS delaying commit events when required. Thus, enforcing CO does not require aborting more transactions than a minimal set of aborted transactions required to comply with SER. This fact is also supported by the CO Commit Delay Lemma (lemma 3.1).

Practically, any scheduler may comply with CO without changing its resource access strategy by delaying commit events when required (mimicking a CO TTS). A similar result can be derived for recoverable, serializable histories.

When a scheduler delegates the commit decision, the following statements about recoverability hold true:

Theorem 4.3 - The Recoverability Inheritance Theorem

Let a system consist of a scheduler that guarantees *recoverability (cascadelessness, strictness)* and some *component* to which it delegates the commit decisions on all unaborted transactions. Let the scheduler vote YES on a transaction when it can be committed (by the scheduler) without violating *recoverability (cascadelessness, strictness).* Then the above system *guarantees recoverability (cascadelessness, strictness* respectively*)* as well.

Proof:

Since the decision notification can arrive any time after voting the scheduler has to guarantee the following condition in order to prevent a possible recoverability violation (see a definition of recoverability in section 2.3 above):

(T_2 reads from T_1) implies ($e_1 < y_2$ and ($e_1 = a$ implies $e_2 = a$))

Thus, if (T_2 reads from T_1) the scheduler does not vote YES on T_2 before T_1 is decided ($e_1 < y_2$) and hence $e_1 < e_2$ follows by rules CDD2,3. Since e_2 follows e_1, if $e_1 = a$ then (by the above condition) the scheduler can and has to enforce ($e_1 = a$ implies $e_2 = a$) by aborting T_2.

Thus, (T_2 reads from T_1) implies ($e_1 < e_2$ and ($e_1 = a$ implies $e_2 = a$))

and recoverability is guaranteed. Note that CD^3C for T_1 and T_2 is maintained.

The cases ACA, ST are straightforward.

§

As a consequence of theorem 4.3 we conclude:

Corollary 4.4

If guaranteeing both CO and recoverability is required, a CORCO is necessary to be combined with a scheduler only if the scheduler does not guarantee recoverability by itself. Otherwise a COCO is sufficient.

Remark:

Note that if the combined scheduler above is S-S2PL based, then the USG of the respective CO TTS does not have any edges (no conflict with an undecided transaction can be generated). This means that no aborts by the CO TTS are needed, as one can expect, and the entire CO TTS is unnecessary. This is an extreme case. Other scheduler types may induce other properties of the respective USGs.

If a COCO is combined with a scheduler that guarantees strictness (ST), then its USG does not reflect *wr* and *ww* conflicts (it reflects *rw* conflicts only).

If a COCO is combined with a scheduler that guarantees cascadelessness (ACA), then its USG does not reflect *wrf* conflicts.

Theorem 4.4 - The Recoverability Enforcement Condition

Let a system consist of a scheduler and some *component* to which the scheduler delegates the commit decisions on all unaborted transactions.

Then

- Guaranteeing CD^3C for any T_1 and T_2 such that T_2 is in a *read-from (wrf)* conflict with T_1 is a *necessary* condition for the system to *guarantee* recoverability.

- Guaranteeing CD^3C for any T_1 and T_2 such that T_2 is in a *wr* or *ww* conflict with T_1 is a *necessary* condition for the system to *guarantee* strictness (ST).

Proof:

(i) Suppose that recoverability is guaranteed and that CD^3C is not guaranteed for all T_1, T_2 such that T_2 has read from T_1. Thus there may exist T_1 and T_2 such that T_2 reads from T_1 , where the scheduler has voted YES on both transactions before T_1 is decided, violating CD^3C. Now suppose that T_1 is aborted by the deciding component and then T_2 is committed. This is a violation of recoverability, contrary to the assumption that the system guarantees recoverability.

(ii) Suppose that strictness is guaranteed. Let $w_1[x]$, $p_2[x]$ be conflicting operations of T_1, T_2 , respectively, where $p_2[x]$ is either a read or write operation. Due to strictness, $e_1 < p_2[x]$ (see section 2.3.2). Since $p_2[x] < y_2$ by CDD1 (see definitin 2.2), also $e_1 < y_2$ follows, which is CD^3C.

§

4.5 Locking based Strict CO (SCO)

Strict CO (SCO = ST∩CO) is an interesting special case of CO, since the recovery of most existing commercial database systems is based on strictness. More concurrency can be achieved, if S-S2PL is replaced with SCO, since S-S2PL blocks access to both read and written resources (by read and write locks), while SCO blocks access to written resources only. Thus, while a read resource is available to be written by another transaction, when enforcing SCO, it is locked until the reading transaction's end if S-S2PL is enforced.

SCO is guaranteed by the following protocol:

Algorithm 4.3 - The Strict-CO (SCO) Algorithm

- (Write) locks are applied to resources that are being modified.
- Locks applied on behalf of a transaction are released only after the transaction has ended (this condition guarantees strictness (ST)).
- If transaction T_2 is in a *rw* conflict with a committed transaction T_1 , then T_2 is not committed before T_1 does. (This is the CO rule, enforced by the CO algorithm (algorithm 4.1) above. Note, that in this case only *rw* conflicts are reflected in the USG, the CO algorithm's data structure; see the remark following corollary 4.4).

§

Like other locking protocols, the above SCO protocol can be implemented through multi granularity locking schemes ([Gray 78]). Since strictness is an inherently-blocking property, the protocol is subject to deadlocks (like S-S2PL mechanisms). Both SCO and S-S2PL based systems may use the same recovery procedures, that are based on strictness (see [Bern 87], [Gray 78]).

SCO can coexist with S-S2PL, i.e., also read-locks can be applied selectively to resources when SCO is enforced, and when read-locks may prevent aborts. This can happen when repeated read operations of a resource, applied by a transaction, are separated by a write operation of the same resource, issued by another transaction. A read lock, applied before the first read and released immediately after the last read, prevents this possibility. Thus, a mixed

SCO - S-S2PL protocol may be advantageous if the probability of such situations is high. Similar considerations apply when a resource is being read for later writing it.

Example 4.3 Blocking situations in SCO versus S-S2PL

The following scenario demonstrates the cases when *commit-blocking* SCO (see definition 3.2) provides a shorter response time than S-S2PL. SCO differs from S-S2PL in read-write (*rw*) conflict situations only. In all other conflict situations (*wr*, *ww*) it has the same blocking behavior as
S-S2PL.

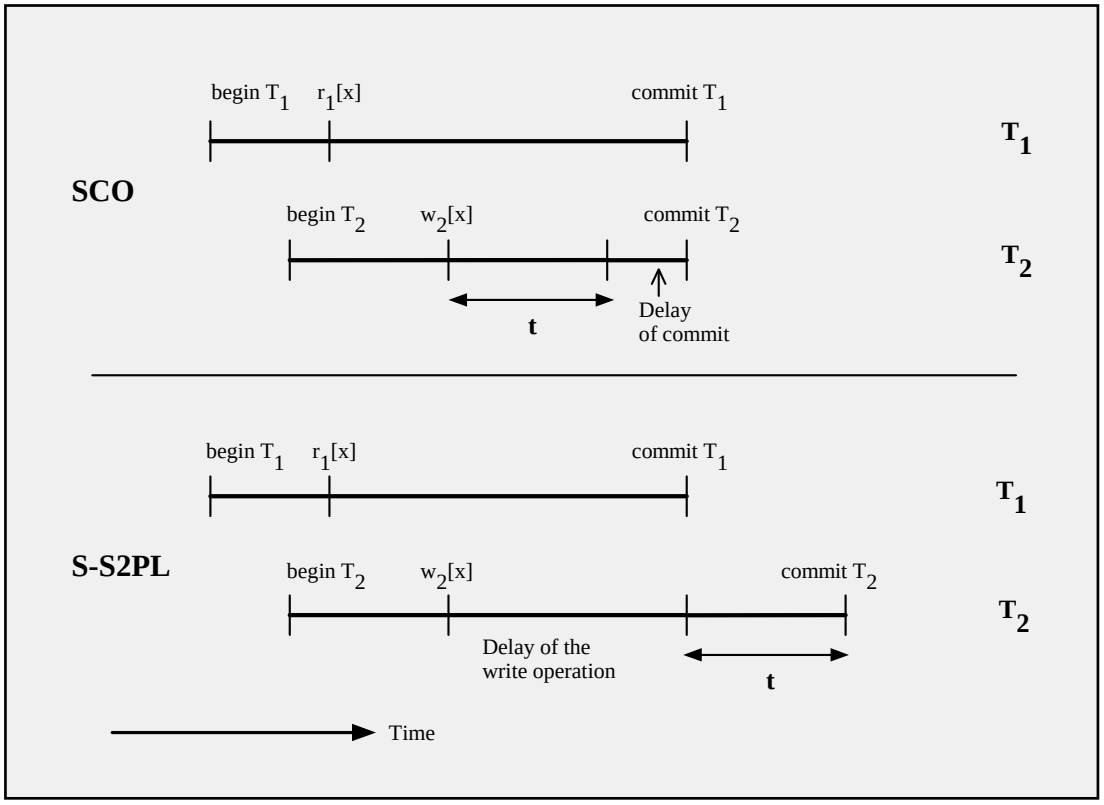


Figure 4.5 : Read-write conflict situation - SCO vs. S-S2PL

For SCO the completion time of transaction T_2 is approximately **t** time units less than that for S-S2PL (assuming that computing resources are available to support parallel executions). Thus, the average response time for SCO is better. Under the conditions of bounded multiprogramming level, reduced response time translates to increased throughput. Note that no commit delay is required if T_2 commits before T_1.

§

Both S-S2PL and commit-blocking SCO are special cases of locking with *Ordered Sharing* (OS), defined in [Agra 91]. OS generalizes 2PL by modifying the blocking semantics of read and write locks in eight different variations (using eight locking schemes ("tables"), $\tau_1,...,\tau_8$, where τ_1 corresponds to 2PL). Like 2PL, also OS is two phased, and uses the end of phase 1 as a serialization point (i.e., the order of any conflicting operations matches the order of these events in respective transactions). When all locks are released at transaction end, OS reduces to commit-

blocking CO ([Agra 92] redifines this special case of OS in terms of *commitment ordering*). Since practically the end of phase 1 cannot be detected when locks are dynamically applied during a transaction, commit-blocking CO is an important subclass of OS. (in analogy, S-S2PL is an important subclass of 2PL.) For this special case [Agra 91] provides simulations result: A τ_1 based protocol generates S-S2PL histories; τ_3 based histories fall into the SCO class; τ_6 based into the CO∩ACA class; and τ_8 based into the CO∩REC class (all with commit-blocking CO). The simulation results shown for S-S2PL and SCO (OS with τ_1 and τ_3, respectively) are consistent with the observations in example 4.3 above, and provide further insight regarding the performance of SCO versus S-S2PL:

- SCO provides comparable or better throughput than S-S2PL for all the experiments.
- While increasing multi-programming level (MPL), when threshing occurs for S-S2PL due to data contention (caused by a high conflict rete), no data contention and threshing is shown for SCO, even for considerably higher MPL levels.
- As a result of the above, when throughput of S-S2PL drops due to data contention, no drop is shown for SCO. Some experiments show throughput advantage of SCO over S-S2PL of approximately 100%.
- The performance advantage of SCO over S-S2PL increases as the number of processing resources (level of parallelism) increases.

More details on SCO, on its comparison with S-S2PL, and on the migration from S-S2PL to SCO can be found in [Raz 92a].

4.6 Timestamp based CO scheduling

Timestamp Ordering (*TO*; see [Bern 87], [Lome 90]) concurrency control mechanisms provide serializability and are based on a timestamp $ts(T_i)$ (e.g., a real number) associated with each transaction T_i. Timestamps are distinct and totally ordered. The most general *Timestamp Ordering rule (axiom)* is the following:

- **TO**

  For any two conflicting operations $p_1[x]$, $q_2[x]$ of any *committed* transactions T_1, T_2, respectively, $ts(T_1) < ts(T_2)$ implies $p_1[x] < q_2[x]$.

  Formally:

  $(e_1=c \text{ and } e_2=c \text{ and } ts(T_1) < ts(T_2) \text{ and } (p_1[x], q_2[x] \text{ conflicting})) \text{ implies } p_1[x] < q_2[x]$

The TO rule defines a noninherently-blocking property (since it can be enforced by aborting either T_1 or T_2 *after* all their operations have been issued, if required), and provides the basis for optimistic (e.g., certifying the condition above when a transaction ends), as well as blocking TO based concurrency control mechanisms.

Let TO be the class of histories generated by a TO mechanism. This class is exactly the class SER.

Theorem 4.5

TO = SER

Proof:

(i) Let a history H be in TO. Suppose that it is not in SER. Then, by theorem 2.1 CSG(H) has a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ (without loss of generality). By the definition of CSG(H), T_{i+1} is in a conflict with T_i , i=1,...,n-1. Thus, by the TO rule, $ts(T_i) < ts(T_{i+1})$, and by induction $ts(T_1) < ts(T_n)$. However, T_1 is in a conflict with T_n, which implies $ts(T_n) < ts(T_1)$. A contradiction.

(ii) Let H be in SER. Thus, its commit projection is equivalent to some serial history H'. Suppose that H is defined over a set of m transactions, and H' over a set of m' ≤ m. Let S be any size m (sorted) sequence of timestamps. Match any size m' subsequence of S with the transaction sequence H' (H' is serial). Match arbitrarily the remaining s-s' timestamps in S with the uncommitted transactions in H. Clearly, the match for H' obeys the rule TO. Since the conflicts among operations of committed transactions are identical for H and H', TO is maintained for H if and only if it is maintained for H'. Thus, H can be properly matched with any given (long enough) timestamp sequence, and H is in TO. To conclude the proof construct a TO based scheduler that generates the timestamp sequence S, schedules operations according to H, matches transactions to timestamps as described above, and certifies the order before transaction end. By the construction above, clearly the committed transactions in H can be committed by the scheduler. Let the scheduler commit these transactions and abort the rest. Thus this TO scheduler generates H. (Comment: The idea of using an equivalent serial history for this part of the proof is from [Bern 90].)

§

In some cases the more restrictive *Blocking Timestamp Ordering rule (axiom)* is applied:

- **BTO**
  For any two conflicting operations $p_1[x]$, $q_2[x]$ of *any* transactions T_1, T_2, respectively,
  $ts(T_1) < ts(T_2)$ implies $p_1[x] < q_2[x]$.

The BTO rule requires that conflicting operations are scheduled according to the timestamps order regardless of whether the transaction is committed. Typically, this requirement is met by assigning a timestamp to a transaction when it begins, and aborting transactions that cannot meet the condition. The class BTO, of all histories generated by a BTO mechanism, is a proper subset of TO. BTO is incomparable with classes other than SER, introduced in section 2.

CO is a proper subset of TO (since TO=SER, and SER strictly contains CO by theorem 3.1), and can be characterized using timestamps and obeying the following *Timestamp Commitment Ordering rule*:

- **TCO**
  For any two *committed* transactions T_1, T_2 with respective conflicting operations,
  $ts(T_1) < ts(T_2)$ implies $e_1 < e_2$.
  Formally:
  (e_1=c and e_2=c and ($p_1[x]$, $q_2[x]$ conflicting) and $ts(T_1) < ts(T_2)$) implies $e_1 < e_2$

Note that the TO and TCO rules are independent of each other.

The theorem below follows by the definitions of TO, TCO and CO:

Theorem 4.6

A history is in CO if and only if it is generated by a mechanism that obeys both the TO and the TCO rules.

Theorem 4.6 means that if the TCO rule is being enforced by any TO mechanism, then only CO histories are generated. The TCO rule can be enforced by delaying commitment events when necessary to comply with the timestamp order.

The diagram below summarizes the relationships between history classes generated by timestamp-based schedulers:

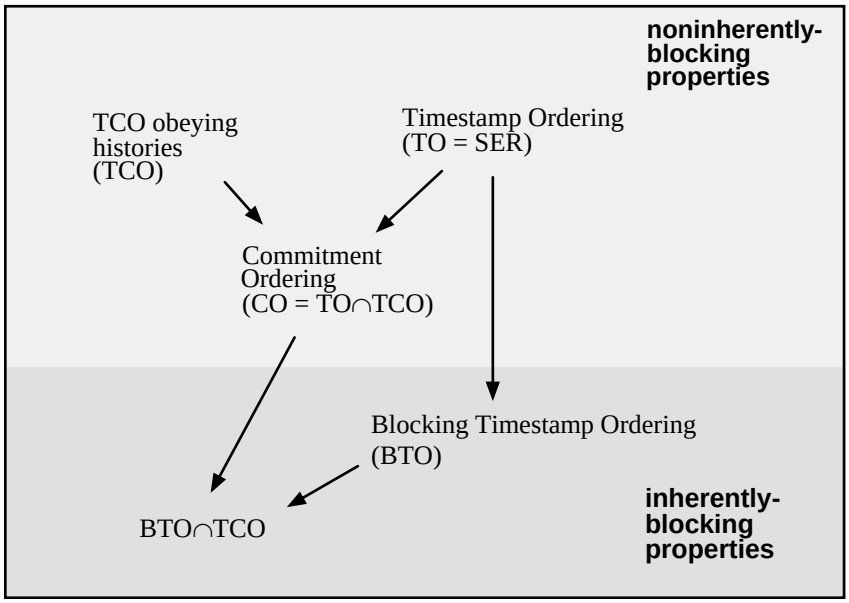


Figure 4.4: Timestamp-based class containment relationships

An arrow from a class A to a class B indicates that class A strictly contains B;
a lack of a directed path between classes means that the classes are incomparable.

A property is *inherently-blocking* if it can be enforced only by blocking transaction's operations until certain events occur in *other* transactions.

4.7 Multi-version based CO scheduling

Multi-version (MV) based concurrency control allows *read-only* transactions (*queries*) to run without blocking *read-write* transactions, or being blocked by them. This section presents a theory for MV concurrency control, where single-version based concurrency control is a special case. Within this theory the single-version CO results generalize in a natural way. With MV scheduling (e.g., see [Bern 87]), a new *resource version* is generated by each write operation of the resource, and the various versions can be read independently. A multi-version (MV) history is a history involving access operations to multi-version resources. Let $r_i[x_j]$ denote a read operation by the transaction T_i of the version of the resource x written by the transaction T_j. Similarly, let $w_i[x_i]$ denote a write

operation of the resource x by the transaction T_i. A multi-version-based CO scheduler should generate *one-copy serializable* (1SER) histories (see [Bern 87]), which provide entities accessing MV resources with the same views (as reflected by resource states) as those provided when serializable histories are generated with single-version resources.

Operation and transaction conflicts, which are the basis for the definition of CO, need to be redefined for the multi-version case[1]. Like in the single-version case, also here *ww* or *wr* transaction conflicts have a correlation with time-precedence between respective operations. However, *rw* conflicts are determined solely by the version read, regardless of the time of reading, i.e., a *rw* conflict can occur even if the read operation takes place *after* the respective write operation. This change requires some modification of the rules HIS2 and HIS3 in the definition of *history* (see section 2.1).

Conflicting are redefined as follows:

If T_i and T_j are in T then the operation of the following pairs are conflicting (if exist):

- $w_i[x_i]$, $r_j[x_i]$, $i \neq j$.
- $w_i[x_i]$, $r_j[x_l]$, $l \neq i$,, $i \neq j$.
- $w_i[x_i]$, $w_j[x_j]$, $i \neq j$.

The following are the modified history rules (axioms):

- **HIS2**

  If T_i and T_j are in T, then the following precedence relationships exist between conflicting operations:

  - If $r_j[x_i]$, $i \neq j$ exists then $w_i[x_i] < r_j[x_i]$.
  - If $r_j[x_l]$ and $w_i[x_i]$, $l \neq i$,, $i \neq j$ exist, then either $w_i[x_i] < r_j[x_l]$ or $r_j[x_l] < w_i[x_i]$ is true.
  - If $w_i[x_i]$ and $w_j[x_j]$, $i \neq j$ exist, then either $w_i[x_i] < w_j[x_j]$ or $w_j[x_j] < w_i[x_i]$ is true.

- **HIS3**

  Let T_i, T_j be transactions in T, where $e_i = a$. Then, if $r_j[x_i]$ exists, then $r_j[x_i] <_H e_i$
  (After T_i is aborted, x_i is inaccessible, i.e., practically does not exist).

Conflicts between transactions are defined as follows:

Definition 4.1

T_j is *in conflict with* T_i in the following cases:

[1] The notion of operation *conflict* used here is different from that in [Bern 87] which defines write-read pairs only as conflicting. However, the results concerning *serializability* (1SER), as implied by the two approaches, are almost semantically equivalent: The formulation in [Bern 87] does not reflect what we consider conflicts involving unread versions. Ignoring such conflicts is analogous, in the single-version case, to ignoring conflicts involving unread resource states.

- If $r_j[x_i]$, $i \neq j$, exists, or if $r_j[x_l]$ exists, and $w_i[x_i] < w_l[x_l]$, $l \neq i$, $i \neq j$, then T_j is *in a wr (write-read) conflict with* T_i.
  For the special case $r_j[x_i]$, $i \neq j$, T_j *reads from* T_i, or T_j is in a *wrf (read-from)* conflict with T_i.
- If $r_j[x_l]$ exists, and $w_l[x_l] < w_i[x_i]$, $l \neq i$,, $i \neq j$, then T_j is *in a rw (read-write) conflict with* T_i.
- If $w_i[x_i] < w_j[x_j]$, $i \neq j$, then T_j is *in a ww (write-write) conflict with* T_i.

§

In dealing with MV resources the definition of a *transaction* (the TR axioms) remains unchanged. Similarly, we maintain the defenitions of *conflict equivalence*, and *serializability graphs* (SG, CSG, and USG; edges represent conflicts between transactions; see section 2) unchanged. The definition of a *history* is slightly changed as reflected by the new HIS2, HIS3 rules above. Note that a MV resource "behaves" (externalizes the same states) as a single-version resource, if only the last existing version can be read by a transaction. Thus a single-version resource can be viewed/defined as a special case of a MV one, when the condition above is met. This observation motivates the definitions that follow, and leads to a correctness criterion for MV histories that generalizes serializability.

A serial MV history is *one-copy serial (1-serial)*, if for all i, j, and x, if T_i reads x from T_j, then i = j, or T_j is the last transaction preceding T_i that writes x ([Bern 87]).

Definition 4.3

A MV history is *one-copy serializable (1SER)*, if its commit projection is conflict-equivalent with a 1-serial history.

§

Theorem 4.7 provides a criterion for one-copy serializability.

Theorem 4.7

A MV history H is one-copy serializable (1SER) if and only if CSG(H) is cycle free.

Proof outline:
CSG(H) as defined here is similar to the graph MVSG(H,<<) defined in [Bern 87] (see last footnote regarding unread versions). It is shown there that one-copy serializability of H is a necessary and sufficient condition for the acyclicity of MVSG(H,<<). The same proof applies here.

§

Note that theorem 4.7 generalizes theorem 2.1 when single-version resources are viewed as a special case of MV resources (as explained above).
We can now conclude that CO implies 1SER:

Theorem 4.8

1SER $\supset$ CO

Proof:
Repeat the proof steps of theorem 3.1 for MV resources and histories, using theorem 4.7 instead of theorem 2.1.

The 1SER history $w_1[x_1]\ r_2[x_1]\ c_2\ c_1$, which is not in CO, demonstrates the strict containment.

§

The CO algorithm (algorithm 4.1) is affected in the MV case by the possibility that incoming edges be generated in the serializability graph for both ready and committed transactions (this is impossible for the single-version case). The next theorem defines under what conditions the CO algorithm is valid for generating MV histories in CO. The only required restriction is avoiding reading committed resource versions that are older than the last committed version. Note that for MV scheduling the USG, the algorithm's data structure, represents MV conflicts as defined above.

Theorem 4.9

The CO algorithm (algorithm 4.1) generates CO, MV histories only (guarantees CO), if and only if no transaction reads a committed version older than the last committed version of a resource. (However, uncommitted versions may be read.)

Proof:
(i) If the condition above is maintained, then the histories generated are proven to be CO by repeating the proof of theorem 4.1 for MV histories. Thus CO is guaranteed.

(ii) Suppose that the condition above is not always maintained. Then, the following scenario is possible:
Suppose that the USG has a single node T_k. Let x_i be the last committed version of x, and x_j the previous committed version. Suppose that T_k reads versions x_j and z_i, and then is being committed. Let H be the history at that stage. Using definition 4.2, the following conflicts exist:

- T_k is in *wr* conflicts with T_i (reading z_i) and T_j (reading x_j).
- T_i is in a *rw* conflict with T_k, since $r_k[x_j]$ exists, and $w_j[x_j] < w_i[x_i]$.

Since T_i is a committed transaction as well, CSG(H) has the edges $T_k \rightarrow T_i$ and $T_i \rightarrow T_k$. Hence, CSG(H) has a cycle, and by theorem 4.7 H is not in 1SER. Thus, H is not in CO by theorem 4.8, and CO is not guaranteed.

§

The version of the CO algorithm (algorithm 4.1), where no committed resource version earlier than the last committed version is read (i.e., where the condithin of theorem 4.9 is obeyed), is referred to as the *MV CO algorithm.* By theorems 4.8 and 4.9 we conclude corollary 4.5:

Corollary 4.5

The MV CO algorithm generates 1SER histories only (guarantees 1SER).

Remarks:

- Since the *read from* relation, i.e., *wrf* conflicts between transactions, is well defined for MV transactions (definition 4.1), recoverability can be applied to the MV CO algorithm by the same technique used to apply it to the CO algorithm (see algorithm 4.2).

- Note that a same transaction may access both single-version and MV resources. This will generate mixed single-version, MV histories. Desired respective properties of mixed histories are maintained by the CO algorithms as long as the respective definitions of conflicts are implemented, and the condition in theorem 4.9 is enforced for MV resources.

More details on MV CO can be found in [Raz 92b].

5 Multiple Resource Manager environment

5.1 The underlying transaction model

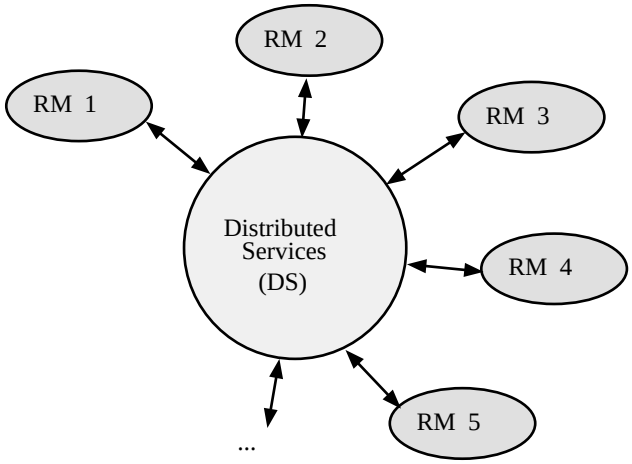


Figure 5.1: A multiple RM environment.
RMs are invoked and coordinated through the DS.

A multi-RM *environment* consists of a set of *autonomous* RMs (more than one), and a *Distributed Services (DS) system* component. A transaction can span any subset of RMs, the *participants* in the transaction.

The DS component provides:

- Application (programs) execution environment
- Application communication services
- Application RM-access services
- Transaction Management services
  (transaction demarcation, RM transaction participation registration, synchronization, atomic commitment, etc.)

No distinction is made between a centralized RM (i.e., confined to a single node) or a distributed one (i.e., executing and accessing resources on several nodes).

Note that the RMs' autonomy implies resource partitioning among the RMs.

The following notation is used:

- Each RM in an environment has an *identifier* (e.g., RM 2).
- Events are qualified by both a transaction's identifier and a RM's identifier (e.g., $w_{3,2}[x]$ means a write operation of resource x by RM 2 on behalf of transaction T_3).

A multi-RM transaction is a generalization of a single RM transaction:

Definition 5.1

A (multi-RM) *transaction* T_i consists of one or more *local subtransactions*[1].
A local subtransaction $T_{i,j}$ accesses *all* the resources under the control of a *participating* RM j, that T_i needs to access, and *only* these resources (i.e., all its events are qualified with j).

A local subtransaction obeys the definition of a transaction and rules TR1, TR2 in section 2.

A local subtransaction has *states* as defined in section 2.

A transaction T_i has an event d_i of *deciding* whether T_i is committed or aborted (see also definition 2.2). d_i is usually distinct from any event $e_{i,j}$ of any local subtransaction $T_{i,j}$ and takes place in the DS component[2].

§

A distinction between an individual RM's history and the global history is required as well. The definition of a (global) history remains unchanged (obeys the definition of a history in section 2).

Definition 5.2

A *local history* is generated by a single RM, and defined over the set of its local subtransactions.

A local history obeys the definition of a history in section 2.

Notation: H_i is the history generated by RM i .

§

It is assumed that an *atomic commitment* (AC) protocol is applied to guarantee *atomicity* in the distributed environment. An AC protocol implements the following general scheme each time a transaction is decided:

- **AC**
  Each participating RM votes either YES (delegates) or NO (also absence of a vote within a time limit may be considered NO) after its respective local subtransaction has reached the ready state, or votes NO if unable to reach the ready state. The transaction is committed by all the RMs if all have voted YES. Otherwise it is aborted by all RMs.

Remarks:

- The YES vote is an obligation to end the local subtransaction (commit or abort) as decided by the AC protocol. After voting YES, a RM cannot affect the decision.
- After voting NO, a local subtransaction may be aborted immediately (thus a NO vote may be represented by the event $e_{i,j} = a$).
- Note that 2PC ([Gray 78], [Lamp 76]) is a special case of AC.

[1] Local subtransactions reflect transaction partitioning over RMs, and are independent of a possible explicit transaction's partitioning into nested subtransactions by an application.

[2] A *local* transaction is a transaction that consists of a single local subtransaction; a *global* transaction consists of two or more. Although a local transaction T_i can be decided locally by some RM j (and not necessarily in the DS), assume for convenience that $y_{i,j}$, d_i exist and obey CDD.

- The AC protocol type used determines under what failure and recovery conditions atomicity is guaranteed (e.g., see [Bern 87]). No specific AC protocols are dealt with here.

AC enforces the following *atomic commitment rules (axioms)* in addition to the rules CDD (see section 2.5):

- **AC1**
  If $d_i = c$ then $y_{i,j}$ exists and $y_{i,j} < d_i$ for all local subtransaction $T_{i,j}$ (i.e., a transaction is decided to be committed only after receiving YES votes for all the local subtransactions).

- **AC2**
  If $d_i = c$ then $d_i < e_{i,j}$ for all local subtransaction $T_{i,j}$ (i.e., all local subtransactions are committed only after the AC protocol has decided to commit the transaction).

For environments that implement AC we conclude the following:

Lemma 5.1 - The Commit Fusion Lemma

Let T_1 and T_2 be any transactions decided via an AC protocol.
If $d_1 = c$ then

- event $< d_1$ if and only if event $< y_{1,j}$ for *some* j, for any event distinct from $y_{1,j}$ and d_1.
- $d_1 <$ event if and only if $e_{1,j} <$ event for *some* j, for any event distinct from $e_{1,j}$ and d_1.
- If d_2 exists then CD^3C for $T_{1,j}$ and $T_{2,j}$ (i.e. $e_{1,j} < y_{2,j}$) for some participant j is a necessary and sufficient condition for $d_1 < d_2$ (compare with theorem 2.3)

Thus, if $d_i = c$ then d_i and all the events $y_{i,j}$ and $e_{i,j}$, for every participating RM j, can be fused together into a single event, without affecting precedence relationships among other events.

Proof:

Follows by the rules CDD and AC.

§

The *commitment fusion lemma* allows in many situations a simplified transaction model to be used. This model ignores the AC mechanism and assumes that a (committed) multi RM transaction (like a single RM transaction) has a single commitment event. Note that this assumption is not valid for the abort events in a multi-RM transaction.

Example 5.1

The following two transactions both access resources x and y.
x, y are under the control of RMs 1, 2 respectively.

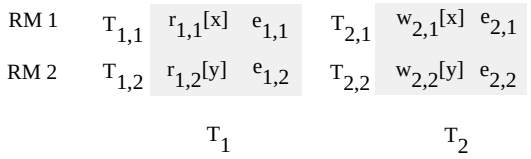


Figure 5.2: T_1 and T_2 and their local subtransactions.

The RMs generate the following histories ($y_{i,j}$ events are omitted):

RM 1: H_1 $r_{1,1}[x]$ $w_{2,1}[x]$ $c_{2,1}$ $c_{1,1}$

RM 2: H_2 $w_{2,2}[y]$ $c_{2,2}$ $r_{1,2}[y]$ $c_{1,2}$

Figure 5.3: The local histories H_1 and H_2.

Note that the history H_1 violates CO, which results in a (global) serializability violation.

The respective global history H is the following:

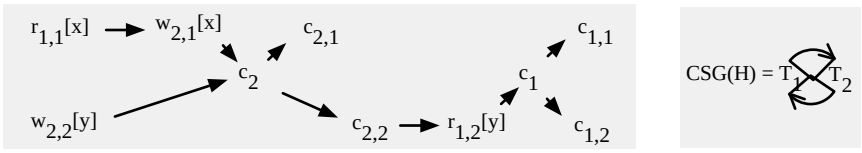


Figure 5.4: The history H and its CSG.

Since the transactions are committed, end-transaction events can be fused with respective decision events (by the fusion lemma).

§

5.2 Local and global properties

This section examines the relationship between properties of histories generated by the individual RMs and the global history generated in the environment. History properties are redefined in a way that allows definitions and results for single RM histories to be used in the multi-RM case.

Definition 5.3 - Global Properties

Let X be a history property, well defined for single RM's histories. Let X', the *associated property of* X, be a (global) history property defined by a modified definition of X, where every event of ending a *committed* transaction (e_i) in the definition of X is replaced with the respective commit-decision event (d_i).

A global history H has property X (is in X) if property X' is well defined and H has property X' (i.e., H obeys the definition of X').

§

Remarks:

- Let X be any history property. In what follows, X also denotes the class of global histories with property X.
- Note that if a property X does not impose constraints on the order of events e_i, then the definitions of X and X' are identical (e.g., serializability).
- Note that when the fusion lemma (5.1) is implemented, the properties X and X' are identical.

Example 5.2

CO', the associated property of CO, is defined as follows:

A history is in CO' if for any conflicting operations $p_{1,j}[x]$, $q_{2,j}[x]$ of any *committed* transactions T_1, T_2 respectively , $p_{1,j}[x] < q_{2,j}[x]$ implies $d_1 < d_2$.

Thus if a (global) history is in CO' , it is also in CO by definition 5.3.

§

The following is a representative example for a result of definition 5.3 :

Corollary 5.1

SER $\supset$ CO (i.e., theorem 3.1 is valid also for global history classes).

The other class containment releationships described in section 3 follow similarly for global history classes.

By lemma 5.1 and the definition of CO we also conclude the following:

Theorem 5.1 - The Global CO Enforcement Condition

- Let H be a global history. Then CD^3C for any *committed* $T_{1,j}$ and $T_{2,j}$ in H (i.e., $e_{1,j} < y_{2,j}$), such that $T_{2,j}$ is in a conflict with $T_{1,j}$, is a *necessary* and *sufficient* condition for H to be in CO.

and thus

- Guaranteeing CD^3C for any $T_{1,j}$ and $T_{2,j}$, such that $T_{2,j}$ is in a conflict with $T_{1,j}$, and $y_{1,j}$ and $y_{2,j}$ already exist, is a *necessary* and *sufficient* condition for guaranteeing CO.
  (Without guaranteeing CD^3C, CO may be violated, and vise versa.)

We now define properties of global histories that reflect properties of their respective local histories:

Definition 5.4 - Local-X

Let H be global history generated in some environment, and H_j its respective local history generated by RM j in the environment.

H'_j, the *augmented* history of a local history H_j , is a history defined over the local subtransactions of RM j, where each local subtransaction $T_{i,j}$ is augmented with the event d_i.

Let X be a history property, well defined for single RM histories, and X' the associated property of X (see def. 5.3). H is in *Local-X* (is *locally* X) if H'_j of every RM j in the environment is in X',

§

Usually (e.g., for all the history properties that we have explicitly defined) Local-X $\supseteq$ X , i.e., if a global history is in X, it is in Local-X. In particular, the following relationships exist as well:

Theorem 5.2

Local-X = X (i.e., a (global) history is in X if and only if it is in Local-X),
where X is any of the following properties:

- REC, ACA, ST, CO, S-S2PL

Proof outline:

The theorem follows by the definitions of global properties (5.3), Local-X (5.4), the commitment fusion lemma (5.1), the RMs' resource partitioning, and the definitions of REC, ACA, ST, CO and S-S2PL.
A proof is demonstrated below for the case X = CO (without using the fusion lemma; it is trivial when the lemma's consequences are considered):

Let H be a global history and H_j its respective local history of RM j, and let H'_j be the augmented history of H_j. With out loss of generality, let T_1 and T_2 be committed transactions in H.
Let H be in CO. Suppose that for some RM j, $T_{2,j}$ is in a conflict with $T_{1,j}$. This means that T_2 is in a conflict with T_1, which implies $d_1 < d_2$ (CO and definition 5.3). Thus for every RM j, H'_j is in CO' (see definition 5.4) and H is in Local-CO.
Now let H be in Local-CO. If T_2 is in a conflict with T_1, it means that for some RM j, $T_{2,j}$ is in a conflict with $T_{1,j}$. Since H'_j is in CO' (Local-CO; definition 5.4) this implies that $d_1 < d_2$. Thus H is in CO (definition 5.3).

§

Theorem 5.3

Local-X $\supset$ X (i.e., being in Local-X does not imply that a history is in X),
where X is any of the following properties:

- SER, 2PL[1], S2PL

Proof:

Local-X $\supseteq$ X is true for SER (using theorem 2.1) and 2PL (follows by definition).
It is also true for strictness:
If H is a global history in ST then ($w_{i,j}[x] < p_{k,j}[x]$ implies $d_i < p_{k,j}[x]$) for any relevant RM j by definition 5.3.

[1] However, for the respective special cases of 2PL and S2PL, where for each transaction the end of phase 1 is synchronized across all the participating RMs, an equality exists, rather than the strict containment.

Thus by definition 5.4 H is in Local-ST.
Thus the containment is also true for S2PL = ST∩2PL.

Now let H be the history in example 5.1 above.

The history H is in Local-SER, Local-2PL and Local-S2PL since both H'_1 and H'_2 are in SER', 2PL' and S2PL' (see definitions 5.3, 5.4).

However H is not in SER, 2PL or S2PL:

- CSG(H) has a cycle, so, by theorem 2.1 H is not in SER .
- If it is in 2PL or S2PL, it is also in SER. A contradiction.

§

5.3 On generating global CO histories: The distributed CO algorithm

This section describes how the Commitment Order Coordinator (COCO) defined in section 4.2 takes part in AC to guarantee global CO histories. (The CORCO, described in section 4.3, is handled similarly.) More implementation oriented details can be found in [Raz 91a].

In a multi-RM environment that implements AC, a COCO typically receives a request via an AC protocol to commit some transaction T in the USG. If the COCO can commit the transaction, it votes YES on it via AC, which is an obligation to either commit or abort according to the decision reached by the AC protocol. Later, after being notified of the decision, if T is committed, all transactions in $ABORT_{CO}(T)$ need to be aborted (by algorithm 4.1). Thus the COCO (say, of RM i) has to delay its YES vote on T, if it has voted YES on any transaction in $ABORT_{CO}(T)$ (CD^3C by theorem 5.1). By similar arguments the COCO cannot vote YES on T, if T is in $ABORT_{CO}(T')$ for some T' on which it has already voted YES. When YES vote on T is possible, the COCO may either choose to do so immediately upon being requested (the *nonblocking without delays* approach), or to delay the voting for a given, predetermined amount of time (*nonblocking with delays*). During the delay the set $ABORT_{CO}(T)$ may become smaller or empty, since its members may be decided and removed from the USG, and since $ABORT_{CO}(T)$ cannot increase after T has reached the ready state. Instead of immediately voting, or delaying the voting for a given amount of time (which may still result in aborts) the COCO can *block* the voting on T until all transactions in $ABORT_{CO}(T)$ are decided. However, if another RM in the environment also blocks, this may result in a global deadlock (e.g., if T' is in $ABORT_{CO}(T)$ for one RM, and T is in $ABORT_{CO}(T')$ for another RM). Aborting transactions by *timeout* is a common mechanism for resolving such deadlocks. Controlling the timeout by the AC protocol, rather than aborting independently by the RMs, is preferable for preventing unnecessary aborts. Note that aborting transactions by the COCO is required only if local cycles in its USG are not eliminated by some external entity (e.g., a scheduler that generates a cycle-free local USG or one that uses aborts to resolve local cycles), or if a global cycle (across two or more local USGs) is generated. Since the cycles are generated exclusively by the way the RASs operate and are independent of the commit order, the COCO does not have to abort more transactions than those need to be aborted for serializability violation prevention (also when using any other concurrency control; see also lemma 4.2).

The following is an AC based CO algorithm (CO-AC), which combines algorithm 4.1 with a generic AC protocol. By the arguments given above, the algorithm enforces CO (and thus serializability) globally if executed by each RM in the environment:

Algorithm 5.1 - CO-AC, The Distributed CO Algorithm

Repeat the following steps:

- Select any transaction T in the USG, that meets the following conditions (using any criteria; selecting T that minimizes the cost of aborting the transactions in the set $ABORT_{CO}(T)$, when T is later committed, is desirable):
  - T is in the *ready* state (i.e., has completed processing).
  - T is not in $ABORT_{CO}(T')$ of any T' on which a YES vote has been issued (theorem 5.1).
  - No YES vote has been issued on any transaction in $ABORT_{CO}(T)$ (theorem 5.1).
- If such a T is found, then vote YES on T.
- Later, asynchronously (after receiving a notification about the decision on T), do the following:
  - If T is committed by the AC protocol, commit T and abort all the transactions in the set $ABORT_{CO}(T)$; otherwise (T is aborted by the AC protocol) abort T.
  - Remove T and the (possibly) other aborted transactions from the graph (they do not belong in the USG by definition).

§

Remarks:

- During each iteration the USG should reflect operations' conflicts of all its transactions until T is committed.
- Note that votes and thus commit decisions are serialized only in the case of conflicts between voted upon transactions. Otherwise any voting order and any commit decision order can be applied by the relevant components.
- By the claim of theorem 5.2 on recoverability, a similar algorithm, for globally enforcing both CO and recoverability, can be devised by combining algorithm 4.2 with an atomic commitment protocol. By theorems 4.3 (recoverability inheritance) and 5.2, it is sufficient for a RM that guarantees recoverability to use algorithm 5.1 without violating global recoverability.
- The distributed CO algorithm extends AC protocols, which achieve consensus among RMs on atomicity, to achieving consensus on both atomicity and global CO.

Example 5.3

The behavior of algorithm 5.1 is demonstrated by the two scenarios below:

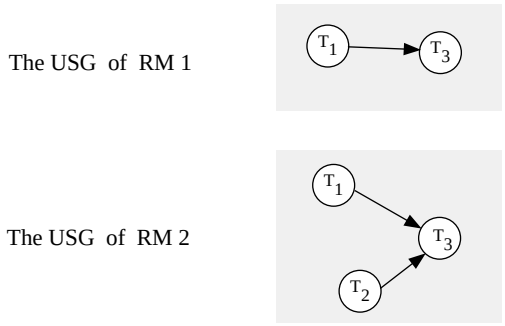


Figure 5.5

To avoid aborting T_1, RM1 first votes YES on T_1, attempting to have it committed before voting YES on T_3 (CD^3C) . Similarly, RM2 votes on T_1 and T_2 before voting on T_3. After both T_1 and T_2 are removed from the USGs, T_3 can be voted on and be committed.

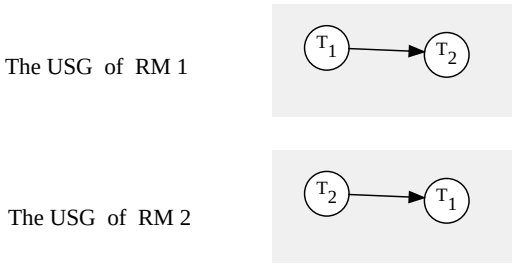


Figure 5.6

RM1 votes YES on T_1 and avoids voting on T_2 until T_1 is decided. RM2 votes YES on T_2 and avoids voting on T_1 until T_2 is decided. This situation results in a deadlock, which can be resolved by the AC protocol aborting either T_1 or T_2 (e.g., by timeout). Global deadlocks may occur even for the nonblocking CO algorithm since the enforced CD^3C condition is blocking.

§

5.4 On CO scheduling and global deadlocks

A *global deadlock* is a deadlock caused by a mutual blocking of two or more local subtransactions of two different transactions at least, in two different RMs at least. Detecting a global deadlock involves RMs exchanging information on blocked transactions. For this reason, in practice, it is common to resolve deadlocks by aborting transactions due to timeout, without actually detecting them (i.e., a timeout condition may be caused by other reasons).

Since CO is noninherently-blocking, it can be implemented in a nonblocking manner (as described in section 4 above). However, even if the schedulers of all the RMs in the environment are nonblocking, global deadlocks can still occur, due to the commit-blocking CD^3C condition (see example 5.3). In this case all the transactions involved with a deadlock are in the ready state. This fact allows to detect deadlocks during atomic commitment of these transactions, while possibly using the AC protocol messages, but violating autonomy, due to piggy-backing auxi-

lary information. If such deadlocks are resolved without detection (e.g., by timeout), autonomy can be maintained. The same is true even when schedulers (one or more) include blocking transaction-termination scheduler components (TTSs), but nonblocking resource-access scheduler components (RASs; see section 4.1) .

If one scheduler, at least, has a blocking RAS (e.g., S-S2PL, SCO, or CO, BTO based), then also active transactions can be involved with a global deadlock (see example 5.4 below). In this case deadlock detection cannot be done during atomic commitment, and additional messages (possibly piggy-backed on AC messages of other transactions) are required for this purpose.

The following example demonstrates a global deadlock that cannot be detected without the cooperation of the involved RMs on this matter:

Example 5.4

Suppose that RM1 uses S-S2PL, RM2 has any CO scheduler, and events are intended to be scheduled as follows:

RM1 ... $w_{1,1}[x]$ e_1 $r_{2,1}[x]$ e_2

RM2 ... $w_{2,2}[y]$ $r_{1,2}[y]$ e_2 e_1

Figure 5.7: A global deadlock situation

RM1 can release the lock on x to execute $r_{2,1}[x]$ only after executing e_1 (S-S2PL). However, RM2 can execute e_1 only after executing e_2 (CD^3C), and e_2 can be executed by RM2 only after RM1 has completed $r_{2,1}[x]$ (by the rules CDD and AC). Thus, T_2 is deadlocked in the active state.

§

6 Guaranteeing global serializability by local Commitment Ordering

This section shows necessary and sufficient conditions for guaranteeing global serializability when the RMs are autonomous, i.e., the only exchanges between them for the purpose of transaction management are those of atomic commitment protocols (with no piggy-backing of any additional concurrency control information; e.g., transaction timestamps).

6.1 Local-CO is sufficient for global serializability

The following is a consequence of theorem 3.1 (corollary 5.1) and theorem 5.2 :

Theorem 6.1

SER $\supset$ Local-CO (i.e., if a history is in Local-CO then it is globally serializable).

Remark: Local-CO is maintained, if all the RMs in the environment use any types (possibly different, e.g., CORCO and S-S2PL based) of CO mechanisms.

6.2 Conditions when Local-CO is necessary to guarantee global serializability

Theorem 6.1 states that local CO is a *sufficient condition* for global serializability. We now use (informally) Knowledge Theory based arguments (see for example [Halp 87], [Hadz 87]) to prove that Local-CO is also *necessary for guaranteeing* global serializability in a multi-RM environment, when the RMs support local serializability and use atomic commitment exchanges only for coordination. (*necessary for guaranteeing* means that otherwise a violation may occur; see definition 2.1.)

The necessity in CO is proven by requiring that each RM avoid committing any transaction that can potentially cause a serializability violation when committed. If it is clear that a transaction remains in such a situation forever (based on knowledge available to the RM locally), the transaction is aborted. We name such a transaction a *permanent risk (PR)*, and later show how to identify it. The PR property is relative to a RM. The above requirement implies that each RM in the environment has to implement the following *commitment strategy* (CS):

- **CS**
  - Starting from a history with no decided transactions, commit any ready transaction via an AC protocol. Every other transaction that is a PR is aborted[1].
  - Repeat (asynchronously when possible; see details below) the following procedure:
    Commit (via AC) any ready transaction, that *cannot cause a serializability violation*, and abort all the PR transactions.

[1] A hidden axiom is assumed, that computing resources are not held unnecessarily. Otherwise, PR transactions can be marked and kept undecided forever. Aborting such transactions and reexecuting them also supports a general concept of *fairness* that requests a transaction's successful completion within a reasonable time interval.

The resulting global histories are proven to be in CO.

Theorem 6.2

- If all the RMs in the environment are autonomous and provide local serializability, then CS is a *necessary* strategy for each RM, in order to guarantee global serializability.
- If CS is implemented by all the RMs, the global histories generated are in Local-CO.

Proof:

The Serializability Theorem (theorem 2.1) implies that the serializability graph provides all the necessary information about serializability. We assume that every RM, say RM i, "knows" its local serializability graph SG_i (it includes all the committed and undecided transactions) and its subgraphs CSG_i (includes all transactions committed by RM i only) and USG_i (includes all undecided transactions only). We also assume (based on AC) that each RM has committed a transaction, if and only if it has voted YES, and "knows" that all other RMs participating in a transaction have voted YES, and will eventually commit it.

The goal for each RM is to guarantee a cycle-free (global) CSG (committed transaction serializability graph), by avoiding any action that may create a global cycle (local cycles in CSG_i are eliminated by RM i, since local serializability is assumed in the theorem).

First, CS is trivially necessary for the following reasons: since a PR transaction remains PR for ever (by definition), it cannot be committed and thus must be aborted to free computing resources. On the other hand, any ready transaction that cannot cause a serializability violation can be committed.

We now need to identify PR transactions, while implementing CS. It is shown that CS implies that each RM executes algorithm 5.1.

Each RM implements CS as follows:

- Base stage:
  Assume that CSG_i does not include any transaction.
  Commit any ready transaction T (via AC).
  Suppose that prior to committing T there is an edge $T' \rightarrow T$ in USG_i. It is possible that there exists an edge $T \rightarrow T'$ in some USG_j of some RM j, $j \neq i$, but RM i, though, may not be able to verify this (due to the autonomy requirement). Without loss of generality suppose that RM i cannot verify this. It means that committing T' later may cause a cycle in CSG. Since committing T cannot be reversed (see transaction state transitions in section 2), no future event can change this situation. Hence T' is a PR (in RM i), and RM i must abort it (by voting NO via AC) upon committing T.

- Inductive stage:
  Suppose that CSG_i includes at least one transaction. We show that no ready transaction can cause a serializability violation, if committed, and hence can be committed (provided that a consensus to commit is reached

by all the participating RMs via AC):

Commit any ready transaction T.

(i) Examine any undecided transactions T' (in USG_i).

Suppose that prior to committing T there is an edge $T' \rightarrow T$ in USG_i. Using again the arguments given for the base stage, T' is a PR, and RM i must abort it. If there is no edge from T' to T, there is no path possibly left from T' to T, after aborting the PR transactions above. Thus no additional T' is a PR and no decision on T' is taken at this stage.

(ii) Examine now any previously committed transaction T" (in CSG_i).

It is impossible to have a path $T \rightarrow ... \rightarrow T"$ in CSG_i or in CSG_j for any RM j, $j \neq i$, since, if this path existed at the stage when T" was committed, it would have been disconnected during that stage, when aborting all the PR transactions (with edges to T"; using (i) above), and since no incoming edges to T" could have been generated after T" has been committed. Hence, only a path $T" \rightarrow ... \rightarrow T$ can exist in CSG_i or in CSG_j for any RM j, $j \neq i$. This means that no cycle in CSG through T and T" can be created, and no T" needs to be aborted (which is impossible since T" is committed, and would fail the strategy).

The arguments above ensure that no ready transaction can cause a serializability violation when committed at the beginning of an inductive stage, as was assumed, and hence (any ready transaction) T could have been committed. Note that committing a transaction can start before the commit process is completed for a previous one (i.e., a concurrent, rather than a sequential implementation of the strategy), as long as CD^3C is maintained for T' and T, where there exists an edge $T' \rightarrow T$ in USG_i (theorem 5.1). Without enforcing CD^3C, a committed transaction may be identified later as a PR in RM i, and cause a serializability violation.

In the CS implementation above, all the PR transactions are identified and aborted at each stage. Examining this implementation we conclude that it results in exactly performing the CO-AC algorithm (algorithm 5.1) in each RM. At the stage of committing T, the set of all PR transactions is exactly the set $ABORT_{CO}(T)$ defined for the algorithm. Hence, by theorem 4.1 every RM involved guarantees CO, and by enforcing CD^3C, also CO' (see definition 5.3). This means that the generated (global) history is in Local-CO (definition 5.4). The only possible deviation from the implementation above is by aborting additional transactions at each stage. Such a deviation still maintains the generated history in Local-CO.

§

Theorems 6.1 and 6.2 imply the following:

Corollary 6.1

Guaranteeing Local-CO is a *necessary* and *sufficient condition* for guaranteeing (global) serializability in an environment of *autonomous* RMs.

Corollary 6.1 has an impact on the correctnes of the *read-only optimization* (e.g., see [Moha 86]) of the Two Phase Commit (2PC) protocol (see also [Raz 95]). Suppose that all the participating RMs in some transaction T_1 use the common S-S2PL protocols for concurrency control. Also suppose that some participating RMs do not modify resources on behalf of T_1. Under the read-only optimization, RMs participating in a transaction, that do not modify resources on behalf of this transaction, are allowed to release their (read) locks upon voting in 2PC (using a special *read-only* voting service). Since due to the read-only vote some (read) locks of T_1 are released before its commit-

ment decision, (global) S-S2PL, CO, and thus serializability may be violated, if another transaction, say T_2 , accesses the released resources and is commit-decided before T_1. Thus the read-only optimization, which is supported by several 2PC implementations (e.g., [LU6.2]}, does not provide system-level guarantees for global serializability, and needs to be carefully utilized by the *applications*, to prevent undesired serializability violations.

Example 6.1

The following scenario of using S-S2PL and the read-only 2PC optimization demonstrates global CO and serializability violation. Upon the read-only vote of RM1 on T_1 the lock on x is released and T_2 is allowed to write x.

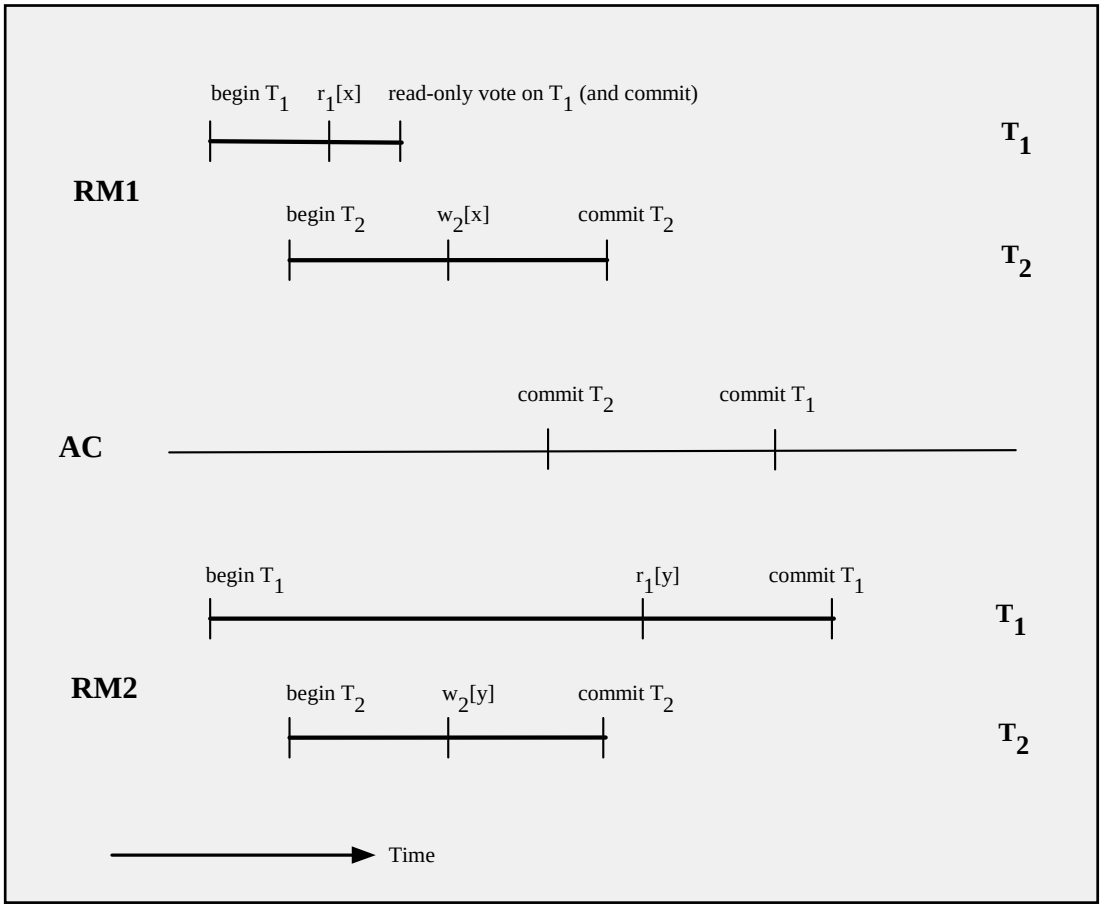


Figure 6.1 : Global serializability violation with 2PC's read-only voting

§

7 Conclusion

This work generalizes a previously known result, that Strong Strict Two Phase Locking (S-S2PL) together with Two Phase Commit (2PC) guarantees global serializability in a multi Resource Manager (RM) environment. The new concept defined here, *Commitment Ordering* (*CO*), provides additional ways to achieve global serializability, through different concurrency control mechanisms, that may provide *deadlock-free* executions. This allows the levels of concurrency to be controlled by local trade-offs between various blocking implementations of CO, with different blocking patterns (e.g., S-S2PL and SCO), which are usually subject to deadlocks, and deadlock-free CO implementations, which are usually subject to cascading aborts (e.g., using the CO-Recoverability algorithm). To guarantee *global serializability*, no services, except those of *atomic commitment*, are required for the coordination of transactions resource access across RMs, if each RM supports CO. It is also shown that guaranteeing CO is *necessary* for guaranteeing global serializability, when the RMs involved are *autonomous* (i.e., when only atomic commitment is used for RM coordination, which is the minimal requirement for RM cooperation on atomic transactions). Since CO does not impose any constraints on resource access patterns, it allows the maximal level of concurrency allowed by any serializable schedule. The only penalty may be the delay of commit of some transactions (which is also true for the special case of S-S2PL).

The relationships between various properties of the histories generated by the individual RMs and respective properties of the respective global history are examined, and assuming that atomic commitment is used, it is shown which properties, CO in particular, are preserved globally when applied locally by the RMs.

Generic CO enforcing mechanisms are described as well, and their behavior in a multi-RM environment is examined. Since CO can be enforced locally in each RM (most existing commercial database systems are S-S2PL based, and already provide CO), no change in existing atomic commitment protocols, services and interfaces is required to utilize the CO solution. Thus, no communication overhead is incurred in this solution. The distributed CO algorithm presented here extends AC protocols (used to achieve consensus among RMs on atomicity) to achieving consensus on both atomicity and global CO, which implies global serializability. Examining the ways CO and recoverability (cascadelessness, strictness) can be combined, we conclude that enforcing CO does not affect already existing recovery procedures of RMs and atomic commitment protocols.

The study presented in this work suggests that CO provides a practical, fully distributed solution for the global serializability problem in a high-performance, distributed transaction-processing environment (see the appendix for implementation-oriented aspects of CO).

Autonomy implies that a RM has no knowledge of whether a transaction is *local*, i.e., confined to the RM, or *global*, i.e., spanning more than one RM. If a RM is coordinated with other RMs via AC protocols only, and in addition can identify its local transactions (e.g., by notifications from applications), it is said to have *extended knowledge autonomy (EKA)*. Since local transactions do not need to be coordinated across RMs via AC protocols, they do not need to obey the CO condition for the purpose of global serializability. Under EKA a more general property, *Extended Commitment Ordering (ECO)*, is necessary to guarantee global serializability ([Raz 91b]). ECO reduces to CO when all the transactions are assumed to be global.

And finally, CO is also a powerful serializability concept in a single-RM environment and the basis for novel locking protocols, as demonstrated by *Strict CO* (SCO; see also [Agra 91], [Agra 92], [Raz 92a]).

Acknowledgments

Many thanks are due to all the people in Digital Equipment Corporation with whom I interacted on this work. Many thanks are due to Joe Twomey for pointing out a mistake in the original formulation of Theorem 2.3..

References

[Agra 90] D. Agrawal, A. El-Abbadi, "Locks with Constrained Sharing", In *Proc. of the Ninth ACM Symposium on Principles of Database Systems*, pp. 85-93, April 1990.

[Agra 91] D. Agrawal, A. El-Abbadi, A.E. Lang, "The Performance of Protocols Based on Locks with Ordered Sharing", in *Proc. of the Seventh Int. Conf. on Data Engineering*, pp. 592-602, April 1991.
A revised version to appear in *IEEE Transactions on Knowledge and Data Engineering.*

[Agra 92] D. Agrawal, A. El-Abbadi, R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases", In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pp. 104-113, June 1992.

[Bern 87] P. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bern 90] P. Bernstein, Private communications.

[Brei 90], Y. Breibart, A. Silberschatz, G. Thompson, "Reliable Transaction Management in a Multidatabase System", in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, New Jersey, June 1990

[Brei 91] Y. Breibart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, A. Silberschatz, "On Rigorous Transaction Scheduling", *IEEE Trans. Soft. Eng.*, Vol 17, No 9, September 1991.

[Brei 92] Y. Breibart, A. Silberschatz, "Strong Recoverability in Multidatabase Systems", in the *Proc. of the Second Int. Workshop on Research Issues in Data Engineering (RIDE)*, Phoenix, Arizona, February 1992.

[DECdtm] J. Johnson, W. Laing, R. Landau, "Transaction Management Support in the VMS Operating System Kernel", *Digital Technical Journal*, Vol 3, no. 1, Winter 1991.

[Elma 90] A. Elmagarmid, W. Du, "A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems", *Proc. of the Sixth Int. Conf. on Data Engineering*, Los Angeles, California, February 1990.

[Eswa 76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., "The Notions of Consistency and Predicate Locks in a Database System", *Comm. ACM* 19(11), pp. 624-633, 1976.

[Garc 88] H. Garcia-Molina, B. Kogan, "Node Autonomy in Distributed Systems", in *Proc. of the IEEE Int. Symp. on Databases in Parallel and Distributed Systems*, pp. 158-166, Austin, Texas, December 1988.

[Geor 91] Dimitrios Georgakopoulos, Marek Rusinkiewicz, Amit Sheth, "On serializability of Multi database Transactions Through Forced Local Conflicts", in *Proc. of the Seventh Int. Conf. on Data Engineering*, Kobe, Japan, April 1991.

[Glig 85] V. Gligor, R. Popescu-Zeletin, "Concurrency Control Issues in Distributed Heterogeneous Database Management Systems", in F. A. Schreiber, W. Litwin editors, *Distributed Data Sharing Systems*, pp. 43-56, North Holland, 1985.

[Gray 78] Gray, J. N., "Notes on Database Operating Systems", *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science 60, pp. 393-481, Springer-Verlag, 1978.

[Hadz 87] Vassos Hadzilacos, "A Knowledge Theoretic Analysis of Atomic Commitment Protocols", *Proc. of the Sixth ACM Symposium on Principles of Database Systems*, pp. 129-134, March 23-25, 1987.

[Hadz 88] Vassos Hadzilacos, "A Theory of Reliability in Database Systems", *Journal of the ACM*, (35)1, pp. 121-145, 1988.

[Halp 87] Joseph Y. Halpern, "Using Reasoning about Knowledge to Analyze Distributed Systems", Research Report RJ 5522 (56421) 3/3/87, Computer Science, IBM Almaden Research Center, San Jose, California, 1987.

[Kung 81] Kung, H. T., Robinson, J. T., "On Optimistic Methods for Concurrency Control", *ACM Tras. on Database Systems* 6(2), pp. 213-226, June 1981.

[Lamp 76] Lampson, B., Sturgis, H., "Crash Recovery in a Distributed Data Storage System", Technical Report, Xerox, Palo Alto Research Center, Palo Alto, California, 1976.

[Litw 89] Litwin, W., H. Tirri, "Flexible Concurrency Control Using Value Date", in *Integration of Information Systems: Bridging Heterogeneous Databases*, ed. A. Gupta, IEEE Press, 1989.

[Lome 90] David Lomet, "Consistent Timestamping for Transactions in Distributed Systems", Technical Report CRL 90/3, Digital Equipment Corporation, Cambridge Research Lab, September 1990.

[LU6.2] System Network Architecture - Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2, SC30-3269-3, International Business Machines Corporation, 1985.

[Moha 86] C. Mohan, B. Lindsey, R. Obermarck, "Transaction Management in the R* Distributed Database Management System", ACM TODS 11(4), pp. 378-396, 1986.

[OSI-CCR] ISO/IEC IS 9804, 9805, JTC1/SC21, *Information Processing Systems - Open Systems Interconnection - Commitment, Concurrency and Recovery service element*, October 1989.

ISO/IEC JTC1/SC21 N4611 Addendum, *CCR Tutorial* - Annex C of ISO 9804, August 1990.

[OSI-DTP] ISO/IEC IS 10026 (1, 2, 3), JTC1/SC21, *Information Processing Systems - Open Systems Interconnection - Distributed Transaction Processing*, 1992.

[OSI-SMO] ISO/IEC DP 10040, JTC1/SC21, *Information Processing Systems - Open Systems Interconnection - System Management Overview*, September 1989.

[Papa 86] Papadimitriou, C. H., *The Theory of Concurrency Control*, Computer Science Press, 1986.

[Pu 88] Calton Pu, "Transactions across Heterogeneous Databases: the Superdatabase Architecture", Technical Report No. CUCS-243-86 (revised June 1988), Department of Computer Science, Columbia University, New York, NY.

[Raz 90] Yoav Raz, "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous, Multi Resource Manager Environment", DEC-TR 841, Digital Equipment Corporation, November 1990; revised April 1992.
An abridged version apeared in the *Proc. of the Eighteenth Int. Conf. on Very Large Data Bases*, pp. 292-312, Vancouver, Canada, August 1992.

[Raz 91a] Yoav Raz, "The Commitment Order Coordinator (COCO) of a Resource Manager, or Architecture for Distributed Commitment Ordering Based Concurrency Control", DEC-TR 843, Digital Equipment Corporation, December 1991, revised April 1992.

[Raz 91b] Yoav Raz, "Extended Commitment Ordering, or Guaranteeing Global Serializability by Applying Commitment Order Selectively to Global Transactions", DEC-TR 842, Digital Equipment Corporation, November 1991, revised April 1992.
An abridged version apeared in the *Proc. of the Twelfth Symp. on Principles of Database Systems*, Washington, DC, May 1992.

[Raz 92a] Yoav Raz, "Locking-Based Strict Commitment Ordering (SCO), or How to Improve Concurrency in Locking-Based Resource Managers - Extended Abstract", DEC-TR 844, Digital Equipment Corporation, January 1992,, revised April 1992.

[Raz 92b] Yoav Raz, "Commitment Ordering Based Concurrency Control for Bridging Single and Multi Version Resources", in the *Proc. of the IEEE Third International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems*,

Viena, Austria, April 1993.
Also available as DEC-TR 853, Digital Equipment Corporation, July 1992.

[Raz 94] Yoav Raz, "Serializability by Commitment Ordering", *Information Processing Letters*, Vol 51, pp. 257-264, 1994.

[Raz 95] Yoav Raz, "The Dynamic Two Phase Commitment (D2PC) Protocol", in George Gottlob and Moshe Y. Vardi (Eds.), *Database Theory - ICDT 95*, pp. 162-176, Lecture Notes in Computer Science, No. 893, Springer-Verlag (The *Proc. of the 5th Int. Conf. on Database Theory*, Prague, Czech Republic, January 1995)

[Rdb/VMS] Lilian Hobs and Ken England, *Rdb/VMS, A Comprehensive Guide*, (section 4.5), Digital Press, 1991.

[Shet 90] Amit Sheth, James Larson, "Federated Database Systems", *ACM Computing Surveys*, Vol. 22, No 3, pp. 183-236, September 1990.

[Silb 91] Avi Silberschatz, Michael Stonebraker, Jeff Ullman, "Database Systems: Achievements and Opportunities", *Communications of the ACM*, Vol. 34, No. 10, October 1991.

[Veij 92] J. Veijalainen, A Wolski, "Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases", in *Proc. of the Eighth Int. Conf. on Data Engineering*, Tempe, Arizona, February 1992.

[Weih 89] William E. Weihl, "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 2, pp. 249-282, April 1989.

[X/Open-DTP] X/Open Company Ltd.[1], DTP X/Open Guide, *Distributed Transaction Processing: Reference Model* (XO/GUIDE/91/020), October 1991.

X/Open Company Ltd., *CAE Specification; Distributed Transaction Processing: The XA Specification*[2] (XO/CAE/91/300), December 1991.

[1] X/Open Company Ltd. is a consortium of vendors that standardizes interfaces for basic services, to support application portability. The X/Open DTP standards are expected to provide the basis for respective IEEE POSIX standards.

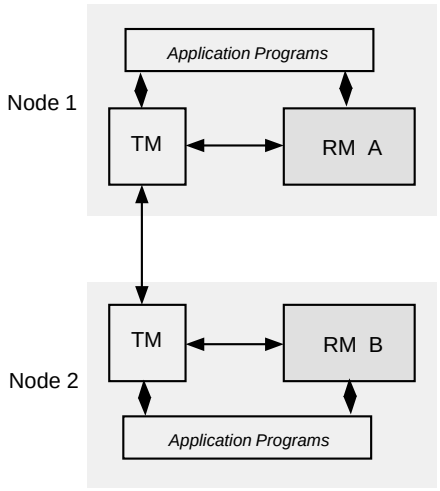
[2] A TM-RM interface.

Appendix - Commitment Ordering Architecture

The *CO architecture* is an extension of the common *Transaction Management architecture* (e.g., [DECdtm], [LU6.2], [X/OPEN-DTP]). The following sections briefly describe this Transaction Management architecture, and then the extension.

A.1 Common Architecture for Transaction Management

Transaction Management is a common name for a collection of services that allow application programs to use the transaction abstraction. Such services typically support transaction demarcation, RM transaction participation registration, transaction termination through *atomic commitment* (AC) etc. A typical architecture for Transaction Management is described below in Figure A.1. The most commonly used AC protocols are variants of the *Two Phase Commit* protocol (2PC; [Gray 78], [Lamp 76]).



RM - Resource Manager

TM - Transaction Manager

Figure A.1 : A common conceptual architecture for implementing the 2PC protocol
2PC is carried out through subsets of the RM-TM interface (e.g., [X/Open-DTP]) and the TM-TM communication protocol (e.g., [OSI-DTP]). Any number of nodes per transaction is allowed. Any number of RMs on a node per transaction, serviced by a local TM is allowed.

The 2PC protocol is typically managed by a distributed *Transaction Manager* (TM) component[1]. A RM participates in the 2PC protocol through an interface with a local (i.e., within a same network node) TM. A common conceptual architecture for 2PC (e.g., see [X/Open-DTP]) involes three generic object types: Application programs, resource managers (RMs), and transaction manager servers (TMs).

We concentrate on a RM's interactions in the 2PC protocol. A common RM-TM interface includes the necessary services to carry out the 2PC protocol. This portion, defined here as the *2PC interface*[2], consists of the following services (The services define here are closely related to the abstract service definition in [OSI-CCR]):

R services: The RM is being invoked

- **R_PREPARE(T)**
  The TM notifies the RM to complete the transaction T. It means that the RM will not receive any additional requests or external data on behalf of transaction T.
- **R_COMMIT(T)**
  The TM notifies the RM to commit transaction T. A prerequisite: the RM has voted YES earlier.
- **R_ABORT(T)**
  The TM notifies the RM (and eventually also all the other RMs involved with T) to abort transaction T.

T services: The TM is being invoked

- **T_READY(T)**
  The RM notifies the TM that it has completed processing transaction T, and it votes YES on T (i.e., it is ready to commit or abort T according to the TM's notification).
- **T_ABORT(T)**
  The RM notifies the TM that it has aborted transaction T (which will result in aborting T by all the RMs involved).

Possible sequences of invocations are defined by the following state transition diagram:

[1] In this work we ignore the *internal* 2PC (or any other AC) protocol of the distributed TM component (i.e., the TM-TM protocol; e.g. see [OSI-DTP]), which does not directly interact with the CO enforcing architected component, the COCO (see section 4 below).

[2] The interfaces described are *abstract*, i.e., they well define the semantics of the related components' interactions, without attempting to define any transfer or invocation syntax on the communication or service boundaries. Though they are common to many architectures and products, they may be packaged and presented there in different ways.

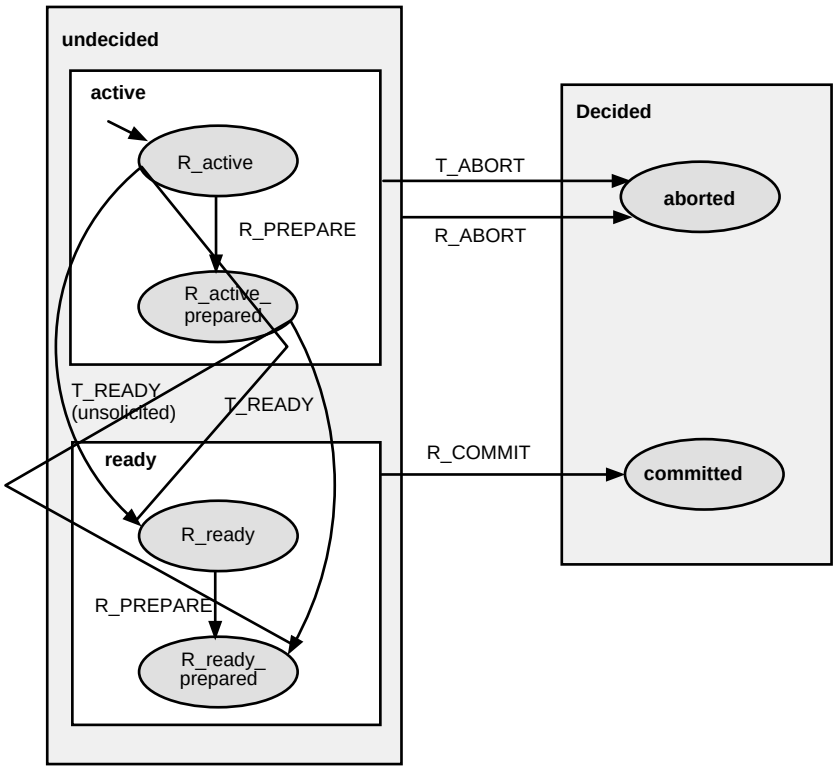


Figure A2: Transaction states and their transitions - The RM's view

Remarks:

- The diagram consists of both "atomic" states and states composed of other states. For example, the state **active** should be interpreted as R_active **or** R_active_prepared, where **or** is the regular logical connective (see more on this type of diagrams, *statecharts*, in [Hare 87]).
- A transaction can be at a single atomic state at a time.
- Though the diagram above is similar to that in Figure 2.1, the definition of the **ready** state here is different: This state is triggered by an explicit T_READY invocation.
- Some TM types do not support the unsolicited T_READY invocation, i.e., an invocation that does not follow an explicit R_PREPARE invocation.

A.2 The Commitment Order Coordinator (COCO) of a Resource Manager

This section defines a software component, the CO Coordinator (COCO). The COCO is defined by its function and interfaces. The interfaces described are *abstract,* i.e., they well define the semantics of the component's interactions with its environment, without attempting to define any transfer or invocation syntax, or to package the invocations in a way that optimizes their total number. These aspects are not dealt with here. Similarly, the appropriate packaging of the COCO in the execution environment (e.g., mapping into processors, servers, processes or thread invocations) is an implementation issue, not dealt with here.

Though for performance considerations the COCO is usually expected to be an integral part of a RM, it is described here as an independent component with a well defined interface with a RM.

The function carried out by the COCO is the CO-Atomic-Commitment (CO-AC) algorithm (algorithm 2.2). The architecture described in what follows is an extension of the generic 2PC architecture described in section 3, and allows multiple concurrent invocations of the COCO, as well as the RM and TM. The COCO acts as the RM's agent in 2PC. It passes *some* of the 2PC message types between the RM and TM through their regular 2PC interface described in section 3. Its added value is enforcing the schedules generated by a RM to be in CO, by *coordinating* the order of commit events with the order of conflicting RM operations. This means that the COCO delays YES votes on behalf of transactions when necessary to comply with CO. If the TM component does not provide a global deadlock resolution mechanism (e.g., by timeout on the 2PC protocol execution duration per a transaction), the COCO can be tuned to timeout delays on voting, and to pay the price of aborting (in $ABORT_{CO}(T)$ sets) more transactions than the minimum necessary, when transactions (respective T) are committed (to prevent potential CO and thus global serializability violations; see algorithm 2.2).

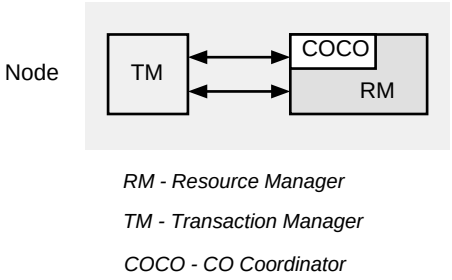


Figure A.3: Positioning the COCO within the common architecture.
The COCO assumes a subset of the 2PC interface.

The subset of the 2PC interface assumed by the COCO comprises the following services: **T_READY**, **T_ABORT**, **R_ABORT** and **R_COMMIT**. All the other services of the TM-RM interface remain there (i.e., they remain direct invocations between the TM and the RM).

Example A.1

Using the COCO in a multi-RM environment:

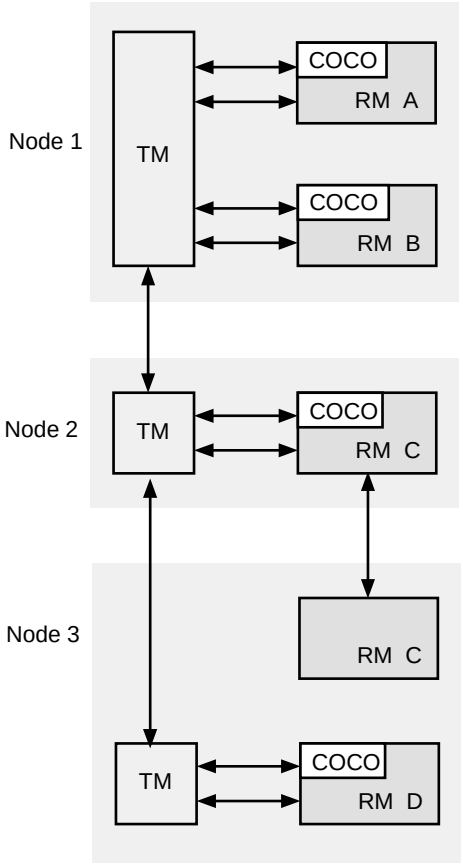


Figure A.4 : A multi RM environment

Each RM (that does not use S-S2PL based CC) has its own COCO component;
RM C is distributed, and uses its own mechanisms to coordinate
concurrency control between its components on different nodes

§

Example A.2

Using the COCO within a distributed RM (multi-node RM) :

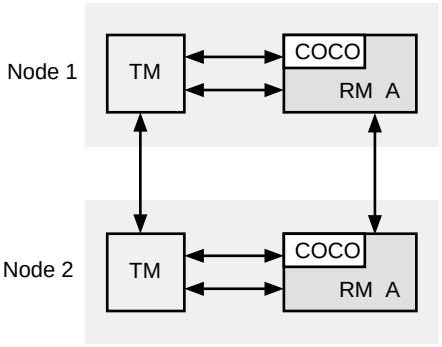


Figure A.5 : RM A is distributed, partitioned, with share-nothing components. The distributed RM's components on different nodes do not share CC information, and implement 2PC (i.e., use TMs) for coordinating AC. Thus, CO may be used for coordinating CC across the RM. In this case a COCO component is needed for each RM's component (that does not implement S-S2PL based CC).

§

A.2.1 Interfaces

The COCO has the following (abstract) interfaces with the TM and RM.

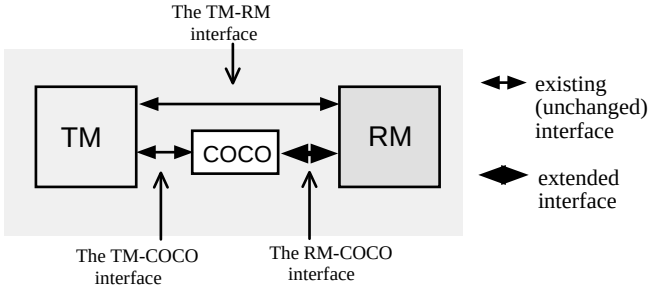


Figure A.6: Abstract interfaces of the COCO

The TM-COCO interface

The TM-COCO interface comprises a subset of the 2PC interface defined in section 3.1 above. Solely for convenience, in this interface we rename some of the R services C_T services, to indicate that these services are invocations of the COCO by the TM. The semantics of the services remain unchanged (see section 3). Thus the interface consists of the following services:

C_T services: The COCO is invoked by the TM

- C_T_COMMIT(T)
- C_T_ABORT(T)

T services: The TM is invoked by the COCO

- T_READY(T)
- T_ABORT(T)

The RM-COCO interface

The RM-COCO interface is a superset of the TM-COCO interface defined above. The additional services are for maintaining the USG (see section 2.3), the COCO's data structure. For convenience we rename here the relevant T services in the 2PC interface C_R services, without changing their semantics (see section 3). Thus the interface consists of the following services:

C_R services: The COCO is invoked by the RM

The original services are the following:

- C_R_READY(T)
- C_R_ABORT(T)

The augmented services are the following:

- **C_R_BEGIN(T)**
  The RM notifies the COCO to establish a node for T in the USG.
- **C_R_CONFLICT(T_1,T_2)**
  Prior to executing an operation of T_2 that generates the conflict with T_1 the RM invokes this service to notify the COCO. If a respective edge from T_1 to T_2 does not exist already in the USG, it is being created. The actual operation of T_2 is executed by the RM only after receiving an acknowledgment from the COCO to guarantee that the USG is updated at that time.

R services: The RM is invoked by the COCO

The original services are the following:

- R_COMMIT(T)
- R_ABORT(T)

The following is an additional service:

- **R_CONFLICT_ACK(T_1,T_2)**
  Conflict acknowledgment. Upon this invocation the COCO is updated already, and the RM can execute the operations in T_2 that cause the respective conflict with T_1.

The TM-RM interface

The *new* TM-RM interface includes all the services (in the original TM-RM interface) that are not assumed by the COCO's interfaces, i.e., all but T_READY, T_ABORT, R_ABORT and R_COMMIT.

Remark:

Since the subset of the 2PC interface assumed by the COCO (vote, commit, abort) is common to all AC protocols, the COCO's interfaces are AC protocol independent. Thus the COCO can be used with any AC protocol.

A.2.2 The CO-AC Algorithm

The CO-AC algorithm (algorithm 2.2) is implemented by combining the COCO's invocations by the RM (C_R services) and the TM (C_T services), with the VOTE algorithm (see below). The VOTE algorithm votes YES on transactions when appropriate, and is being executed continuously by the COCO.

The underlying data structure is the *Undecided transaction Serializability Graph* (USG) defined in section 2.3.

Transaction states and their transitions are described by the state-diagram in Figure 4.5 below. Each transaction (node) in the USG is associated with a single atomic state at a time. States are being changed by the COCO's invocations and the VOTE algorithm. The COCO's invocations are the following:

- C_R_BEGIN, C_R_CONFLICT, C_R_READY, C_R_ABORT;
- C_T_COMMIT, C_T_ABORT.

Remark: Concurrency of invocations and the VOTE algorithm may be handled by applying locks on the USG's nodes where necessary, and executing each invocation of the COCO in uninterrupted mode (critical section). These aspects are implementation specifics, not dealt with here.

A.2.2.1 Invocation sequences

The following diagram describes the allowed invocation sequences in the COCO's interfaces:

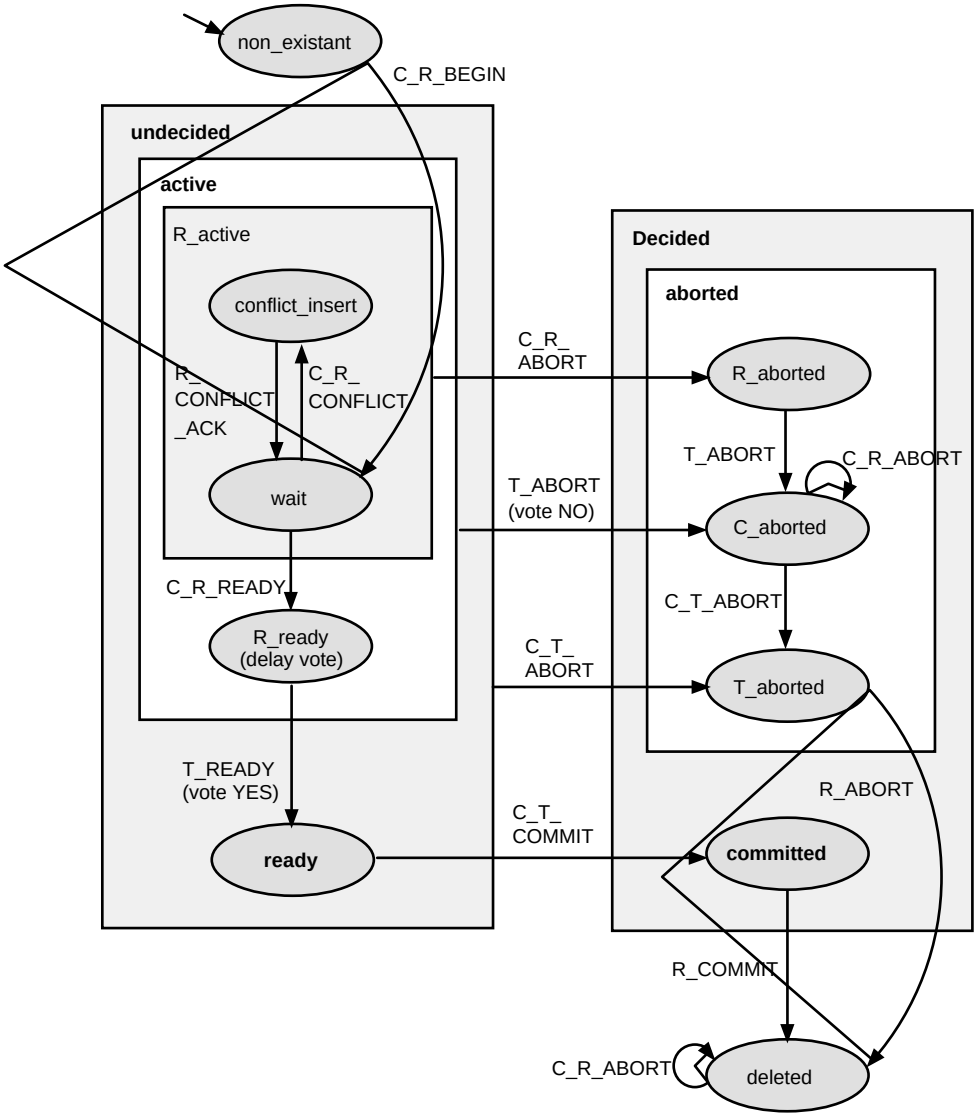


Figure A.7 : Transaction states and their transitions - The COCO's view

A transaction T assumes the *wait* state upon being generated in the USG following a C_R_BEGIN(T) invocation. Upon a C_R_CONFLICT(T',T) invocation, T enters the *R_active* state, and returns to *wait* upon a R_CONFLICT_ACK(T',T) invocation, after the USG has been updated with the respective conflict (one such invocation per USG edge is sufficient). After the RM has voted YES on T by invoking C_R_READY(T), T assumes the *R_ready* state. Up to that YES vote, the RM can abort T by invoking C_R_ABORT(T) (a NO vote by the RM). The composed state *R_active* represents (from the COCO's point of view) the active state of T in the RM. By voting YES on T (using C_R_READY(T)) the RM delegates the voting right to the COCO. Now it is up to the COCO (more specifically, the VOTE algorithm running by the COCO; see below) to decide whethter to vote YES on T (by invoking T_READY(T)), or to later cause it to be aborted (by invoking T_ABORT(T); a NO vote). After

voting YES the COCO cannot abort T anymore. Note that the TM (the AC protocol) can abort T anytime. On the other hand, the RM and COCO can abort only before voting YES. Abort invocations for T, issued by the three components (RM, COCO, TM), may collide, as reflected in figure 4.5.

Remark: The states *conflict_insert, R_aborted T_aborted* and *committed* are transient, i.e., they are intermediate states during the COCO's invocations. For example, upon a C_T_COMMIT(T) invocation T assumes the *committed* state temporarily, and then the state *deleted* (see algorithm 4.1 below).

A.2.2.2 The COCO's invocations

The COCO's invocations maintain the USG, the COCO's data structure (including transactions' states), and execute the CO algorithm (see sections 4, 5). As was noted earlier, concurrent invocations are allowed. The following algorithm is an example for a COCO's invocation.

Algorithm 4.1 - The C_T_COMMIT(T) invocation

Parameter: T

- error condition is raised if T is not in the *ready* state.
- T assumes the *committed* state.
- For every T' in $ABORT_{CO}(T)$
  - error condition is raised if T' is not in *active* or *C_aborted* state.
  - if T' is in the *active* state then
    T_ABORT(T') is invoked (NO vote on T') and
    T' assumes the *C_aborted* state.
- R_COMMIT(T) is invoked.
- T is deleted from the USG and assumes (implicitly) the state *deleted*.
- Exit.

§

A.2.2.3 The VOTE algorithm

The following VOTE algorithm selects transactions to be voted YES on. It is executed continuously by the COCO:

Algorithm 4.2 - The VOTE algorithm

Repeat the following steps:

- Select a transaction (node) T in the USG that obeys the following conditions:
  - T is in the *R_ready* state.
  - T is not in $ABORT_{CO}(T')$ for any T' in the *ready* state (i.e., on which a YES vote has been issued).

- No T' in $ABORT_{CO}(T)$ is in the *ready* state (i.e., no YES vote has been issued on T').
- T is optimal regarding the effects of aborting the transactions in $ABORT_{CO}(T)$ if T is committed (using any optimality criteria, possibly by priorities assigned to each transaction; a priority may be changed dynamically as long as the transaction is in the USG; selecting T with an empty $ABORT_{CO}(T)$ set or one with decided members only (which will be deleted soon) is always preferable, provided that no transaction is blocked undecided too long as a result of this, i.e., *fairness* is maintained).

- If such a T is found, invoke T_READY(T) (vote YES on T).

§

Remark:

The priority of a transaction T, to be voted YES on, may reflect the accumulated cost of the respective local subtransaction. If also the cost of other local subtransactions of T is taken into account, i.e., the priority is based on the total cost of T, this global information should be passed through the TM-COCO interface, and carried through the AC protocol.

Either the TM or RM or possibly the COCO (this option for the COCO is not explicitly shown here) guarantees (by aborting transactions, usually due to timeout) that no transaction is blocked in the *C_ready* state too long, and thus the VOTE algorithm will eventually vote YES on some transaction. In order to reduce the number of unnecessary aborts it is recommended that the above blocked transactions are aborted by the TM, rather than independently by the RMs or their respective COCOs. The reason is that the distributed TM, being the final decision maker on commits, can control aborting transactions sequentially, while unblocking blocked transactions. The RMs (COCOs), on the other hand, cannot coordinate aborts, and may abort concurrently blocked transactions, that otherwise would be later unblocked by other aborts, and complete successfully.

A.3 Extensions of the COCO

A.3.1 Distingushing between local and global transactions

The COCO described above assumes that all the transactions involved are decided by the AC protocol. However, for most existing applications, most[1] of the transactions are *local*, i.e., confined to a single RM, and are decided by that RM, rather than by the AC protocol, outside of the RM. Only transactions that are *global*, i.e., span two or more RMs, need to be decided by the AC protocol. For supporting such funcionality, the COCO has to distinguish between local and global transactions (the COCO still needs, though, all the transactions being reflected in its USG). This can be done by tagging local transactions in both the RM-COCO interface (e.g., in the C_R_BEGIN service) and the COCO's USG (its nodes). For local transactions the action of the COCO voting (T_READY) is replaced by the COCO *committing* (note, however, that the RM still has to vote using C_R_READY). In the spe-

[1] This may change in the future, when transactions over multiple RMs become more pervasive.

cial case when all the transactions are local, the TM-COCO interface is not used in this version of the COCO, and the CO-AC algorithm reduces to the single RM CO algorithm. Typically the RM acquires the knowledge about transactions being local by notifications from the application (programs). Another solution may be applied in an environment where each transaction is known to the distributed TM: The TM may identify local transactions (using a specialized distributed algorithm) and notify the respective RMs.

With the capability to distinguish between local and global transactions the COCO may take advantage of the *Extended Commitment Ordering* (ECO) property ([Raz 91b]). ECO generalizes CO and is based on RMs having the knowledge to make a distinction between local and global transactions. While CO enforces a certain (partial) order of the commit events of all the (committed) transactions in a history, ECO enforces a similar condition on global transactions only. Unlike CO, that being enforced locally it implies also local serializability, ECO *needs* to be accompanied with local serializability in order to guarantee global serializability[1]. ECO reduces to CO when all the transactions are assumed to be global (e.g., if a RM cannot distinguish between local and global transactions). A generic ECO algorithm is described in [Raz 91b]. The enhanced COCO (with tagged local transactions) may execute this less constraining but a little more complicated ECO algorithm. From the TM's point of view, the voting order by the COCO (on global transactions only in this case) is identical for the CO and the ECO algorithms, and in both cases global serializability is maintained.

A.3.2 Distingushing between queries and updaters

Performance may be considerably increased for loads with long *queries* (*read-only* transactions), if different resource access strategies are applied to queries and *updaters* (*read-write* transactions). This requires supporting *multi-version* (MV) resources (e.g., [Bern 87], [Rdb/VMS], [Raz 92b]; see also section 4.7). The COCO generates CO histories also for MV resources, if the notion of a conflict is generalized appropriately (see section 4.7, and more details in [Raz 92b]). For MV resources CO implies the *one-copy-serializability* history property (1SER), which is the correctness criterion for MV resources, corresponding to serializability in the single-version case. The COCO itself (interfaces and algorithm) is unaffected by the generalization of the notion of conflict. This generalization affects only the components that identify conflicts.

Typically for MV based RMs, only updaters use the COCO (and reflected in the USG), while queries may not comply with CO and use specialized resource access strategies, bypassing the COCO
([Raz 92b]).

A.3.3 A COCO that enforces recoverability (CORCO)

The *recoverability* property of histories (e.g., see [Bern 87], [Hadz 88]) is necessary for a RM to correctly recover resources after aborts. It guarantees that no committed transaction has read resource states (data) written by an aborted transaction (corrupted data). The COCO described above relies on the RM to guarantee recoverability, i.e., if the RM guarantees recoverability, also the schedules generated by combining a COCO preserve this property

[1] ECO is the most general property that guarantees global serializability (a necessary condition for global serializability) when applied locally (together with local serializability) by RMs that communicate solely via AC protocols and can identify their local transactions ([Raz 91b]).

(recoverability *inheritance*; see theorem 4.3). When the RM scheduler does not guarantee the recoverability property (which is unlikely for DBSs), the COCO can execute a modified CO algorithm, algorithm 4.2, to guarantee also recoverability by considering *read-from* conflicts and applying cascading aborts when necessary. This requires that read-from conflicts are marked both in the RM-COCO interface (e.g., in the C_R_CONFLICT service) and in the COCO's USG edges.

A.4 Summary

This appendix provides specifications of a software component, the *Commitment Order Coordinator* (COCO). When the COCO's instances are appropriately embedded in a distributed, multi RM, Atomic Commitment (AC) supporting environment, the generated global histories have the *Commitment Ordering* (CO) property. This guarantees global serializability.

The generic COCO's architecture described here has the following properties:

- It allows efficient, fully distributed implementations of CO.
- No change in existing interfaces and protocols of *transaction management services* is required for introducing COCO components in RMs.
- The COCO can be implemented in any RM that uses any type of concurrency control, as well as RMs with no concurrency control at all. This is true for both single-version and multi-version based RMs. The COCO is passive and does not affect the RM's resource access strategy. However, it requires operation conflict notifications. A conflict may be defined by any semantics applicable to the RM.
- It allows both local and global transactions to execute concurrently without any constraint except the CO condition. This constraint does not prohibit any desires resource access strategy, and allows any concurrency control mechanism to be used in each RM.
- Moreover, The CO constraint does not require aborting more transactions than those required to be aborted for serializability violation prevention. The only cost of enforcing CO may be the possible delays in commit decisions, and the overhead of a conflict notification mechanism (if does not exist anyhow; e.g., a lock manager can provide all the required notifications for some CO algorithms). Also transaction commitment is executed concurrently, except for distributed conflicting transactions, which are voted serially.
- Since the only RM communication required by the COCO is that of AC protocol messages (which are required anyhow to support transaction atomicity), no communication overhead is incurred by the COCO solution.
- Since all atomicity recovery aspects are handled by the RMs and TMs (AC protocol), a COCO's implementation only needs to comply with the respective interfaced RM 's and TM's recovery requirements.

The above properties make the COCO architecture a practical solution to the global serializability problem in a heterogeneous, distributed, multi RM, high performance transaction processing environment. Since S-S2PL based RMs (e.g., most existing commercial Database Systems) are CO compliant, they can participate in such multi RM environments without being modified (they do not need a COCO). Rephrasing this, any non S-S2PL based RM can join existing S-S2PL based environments by adding a COCO (if not CO compliant already).