Distributed and Parallel Databases, 11, 239–268, 2002

Dictatorial Transaction Processing: Atomic Commitment Without Veto Right*

MAHA ABDALLAH Maha.Abdallah@prism.uvsq.fr
Laboratoire PRiSM, Université de Versailles, 45, avenue des Etats-Unis, 78035 Versailles, France

RACHID GUERRAOUI Rachid.Guerraoui@epfl.ch
Département de Systèmes de Communication, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland

PHILIPPE PUCHERAL Philippe.Pucheral@prism.uvsq.fr
Laboratoire PRiSM, Université de Versailles, 45, avenue des Etats-Unis, 78035 Versailles, France

Recommended by: M. Tamer Özsu

Abstract. The current standard in governing distributed transaction termination is the so-called *Two-Phase Commit* protocol (2PC). The first phase of 2PC is a *voting* phase, where the participants in the transaction are given an ultimate right to abort that transaction. Giving up that *veto right* from all participants reduces the overhead of the atomic commitment protocol but also imposes some restrictions on the concurrency control and recovery protocols employed by the participants in the transaction.

This paper gives, for the first time, a precise abstract specification of the *Dictatorial Atomic Commitment* (DAC) problem, resulting from removing *veto rights* from the traditional *Atomic Commitment* (AC) problem. We characterize transactional systems that are compatible with that specification in terms of necessary and sufficient conditions on concurrency control and recovery protocols, and discuss the practical impacts of those conditions. From this study, we capitalize on existing protocols that solve the DAC problem, and propose a new protocol that broadens the applicability of dictatorial transaction processing in order to meet the requirements of today's distributed environments. We point out interesting performance tradeoffs, and describe the implementation of our protocol in the context of current transactional standards, initially designed with 2PC in mind.

Keywords: distributed transactions, atomic commitment protocols, dictatorial transaction processing, two-phase commit, one-phase commit, concurrency control, recovery, CORBA, OTS

1. Introduction

In a distributed transactional system, all sites accessed by a transaction (called transaction *participants*) must coordinate their actions so that they either unanimously *commit* or unanimously *abort* that transaction. This is achieved through an *Atomic Commitment Protocol* (ACP) launched at the end of the transaction. The best known of these protocols is the *Two-Phase Commit* (2PC) protocol [10]. Although widely used and de facto standard [18, 24], 2PC has two major drawbacks:

*This work has been partially funded by the CEC under the OpenDREAMS Esprit project n°20843.

- It is considered as quite *inefficient* in terms of both time delay and message complexity. This is mainly due to the number of communication steps and forced log writes needed in order to commit a transaction even in the absence of failures. This not only makes 2PC inadequate to today's highly reliable distributed platforms, but also is particularly unacceptable in advanced and critical applications, such as Supervision and Control Systems' applications (SCS)[1] [1] having strong requirements in terms of *performance.*
- It forces participants in a transaction to externalize a local *prepared state.* The consequence of this is threefold. First, it violates *site autonomy*, precluding the integration of legacy systems [20] in distributed transactions. Second, building a local prepared state can be very costly on data servers hosted by lightweight intelligent devices with very limited resources, such as palmtops, cellular phones, or even smart cards [7]. Third, it leads to the abort of a transaction after it has been successfully processed if any of its participants is unreachable during the voting phase. The impact of this behavior is exacerbated in environments supporting mobile or disconnected computing [7].

As its name indicates, 2PC (and its variations) is made out of two phases. In the first phase, called the *voting* phase, the participants are given an ultimate right to abort the transaction (i.e., the *veto right*), and in the second phase, called the *decision* phase, the participants need to agree on the same decision (*commit* or *abort*). Whereas the *decision* phase is indeed necessary to ensure transaction *atomicity* (otherwise the participants might disagree on the transaction's outcome), one might wonder whether the *voting* phase can (sometimes) be eliminated. This would drastically reduce the cost of commitment (two communication steps together with their associated forced log writes would be gained), and participants would not need to externalize a local prepared state anymore. Roughly speaking, to commit a transaction, a coordinator would simply need to force-write the decision and send one message to the participants.

The idea of *One-Phase Commit* (1PC) is not new: it was informally discussed by Gray in [10, 11] as well as by Stonebraker in [23]. More recently, several 1PC variations have been suggested in the literature, such as *Early Prepare* (EP) [21, 22], *Coordinator Log* (CL) [21, 22] and *Implicit Yes-Vote* (IYV) [3, 4] protocols. Despite their efficiency, 1PC protocols have been completely ignored in the implementation of distributed transactional systems. We believe that the reason for this is due to some (strong) assumptions made by 1PC protocol designers about the underlying transactional systems without any statement on the necessity of those assumptions. This gives the impression, from a practical point of view, that 1PC is just an exotic concept with unrealistic underlying assumptions and, from the theoretical point of view, that 1PC does not make any sense as it contradicts proven lower bounds on the cost of solving the atomic commitment problem in distributed systems [9].

Our primary goal in this paper is to better identify the assumptions under which 1PC can be used. To our knowledge, none of the papers that were devoted to 1PC either defines the abstract properties of the problem that is solved or gives a precise description of the impact of eliminating the *voting* phase on transaction processing.

We first point out the fact that removing the *veto right* from atomic commitment comes down to define an *agreement* problem that is different from the traditional atomic commitment problem solved by a 2PC [5]. We then give an algorithm that solves the resulting

problem, which we call the *Dictatorial Atomic Commitment* (DAC) problem. A crucial feature of this algorithm is that it can be seen as the basic building block around which all existing 1PC variations are designed. The lack of the *veto right* explains why 1PC is actually more efficient than any of the well-known optimized variations of 2PC, such as *Presumed Commit* (PrC) and *Presumed Abort* (PrA) [17].

We next give three conditions that are necessary and sufficient to ensure the correctness of transactional systems with no participant *veto right*: *on-line serializability*, *cascadelessness* and *on-line resiliency*. These conditions are strictly stronger than the usual correctness metrics for transactional systems, namely *serializability*, *recoverability* and *resiliency*, respectively [13]. Unlike *on-line serializability* and *cascadelessness*, *on-line resiliency* is however rarely realistic in practice. We discuss techniques employed by existing 1PC protocols to circumvent the need for this condition by considering *non-classical* atomic commitment schemes that allow participants to delegate part of their transactional responsibilities to the coordinator of the protocol, and we use these schemes to better explain some differences and similarities between *Early Prepare*, *Coordinator Log* and *Implicit-Yes Vote* protocols. We stress the fact that although the existing techniques eliminate the need for *on-line resiliency*, they come at an additional cost that compromise their use in real systems. We then study an adaptation of those techniques and propose a new protocol, named *Coordinator Logical Log* (CLL), which capitalizes on all existing 1PC variations while alleviating their drawbacks, making 1PC indeed realistic and useful in practice.

We finalize our work by showing the way *Coordinator Logical Log* can be integrated into well-known transactional standards, namely OMG's *Object Transaction Service* (OTS) [18], on top of a CORBA architecture [19].[2]

The remainder of the paper is organized as follows. In Section 2, we describe the distributed transaction model we follow throughout the paper. Section 3 defines the atomic commitment problem and briefly recalls the principle of two-phase commit together with its well-known optimizations. In Section 4, we give a precise specification of the abstract problem resulting from removing *veto rights* from atomic commitment, describe a basic 1PC protocol that solves this problem, prove its correctness, and identify the basic assumptions underlying it. Section 5 and Section 6 make an in-depth analysis of the impact of removing *veto rights* from atomic commitment on concurrency control and recovery protocols, respectively. Section 6 extends the results on recovery to compare existing 1PC variations, point out their practical limitations, and describe our CLL protocol and its associated recovery algorithm. In the same section, we compare the performances of CLL with other existing 1PC protocols. Section 7 studies the implementation of CLL in the context of OMG's OTS. Finally, Section 8 summarizes the main contributions of the paper and discusses some interesting extensions to this work.

2. Distributed transaction model

We consider a distributed system composed of a finite set of sites completely connected through a set of communication channels. Each site has a local memory and executes one or more processes. For the sake of simplicity, we assume only one process per site. Processes (sites) communicate with each other by exchanging messages.

At any given time, a process may be *operational* or *down*. While operational, a process is assumed to follow exactly the actions specified by the algorithm it is running. Operational processes may go down due to *crash failures*.

We consider a *crash-recovery* failure model in the sense that a process can be down (crash) and later become operational again. When it does so, we say that the process *recovers*, in which case it executes a specific *recovery protocol*. A process that is down stops all its activities, including sending messages to other processes, until it recovers. Each process has access to a *stable storage* (i.e., that sustains crash failures) in which it maintains information necessary for the recovery protocol. During recovery, a process restores its local state using the information it wrote on stable storage.

Although processes may crash, we assume that communication is reliable in the following sense: if a process P_i sends a message to a process P_k, then unless one of them crashes after the message is sent, the message is eventually received by P_k.[3]

A *distributed transaction* (henceforth called a "transaction") accesses shared objects residing at multiple sites. For each transaction, the set of processes that perform updates on its behalf are called transaction *participants*. The portion of a transaction executed at one participant is called a transaction *branch*. In the following, we assume that each participant ensures the well-known ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*) [8] properties of every transaction branch it executes. We also assume that for every transaction, there is one specific participant, called the transaction *coordinator*, which manages the transaction processing and termination.[4] The coordinator forwards every transaction operation to the participant hosting the object involved by the operation. If a participant succeeds in processing an operation, the participant replies by sending back an *acknowledgment* message; otherwise, the participant aborts the transaction and sends back a *negative acknowledgment*. To conclude a transaction, the coordinator triggers an *Atomic Commitment Protocol* (ACP) whose role is to ensure a consistent termination of the transaction at all participants.

3. Atomic commitment: Background

In this section, we recall some background about the atomic commitment problem, and discuss the basic 2PC protocol together with its optimized variations.

3.1. The atomic commitment problem

The *Atomic Commitment* (AC) problem is an agreement problem that is concerned with bringing all participants in a transaction to agree on a unique outcome (*commit* or *abort*) for that transaction. This problem was formally defined in [5]. Each participant has exactly one of two votes: *yes* or *no*, and can reach exactly one of two decisions: *commit* or *abort*, such that the following properties are satisfied:

AC-Agreement: No two participants reach different decisions.
AC-Validity: *commit* is decided only if all participants vote *yes*, and if all participants vote *yes* and no failures occur, then all participants must decide *commit*.
AC-Integrity: No participant can reverse its decision after it has reached one.

The vote of a participant reflects its ability to commit its transaction branch. A participant votes *yes* only if the local execution of its transaction branch was successful and it is *ready* and *willing* to make its updates permanent even in the presence of failures. This actually means that the participant can locally guarantee the ACID properties of its transaction branch. A *no* vote (or *abort*) indicates that due to some local problems (integrity constraint violation, concurrency control problem, memory fault or storage problem), the participant is not able to guarantee some of the ACID properties of its transaction branch.

An *Atomic Commitment Protocol* (ACP) is an algorithm that satisfies all of the three properties of the AC problem.

3.2. Basic two-phase commit (2PC)

3.2.1. Protocol description. The basic *Two-Phase Commit* (2PC) protocol (together with its variations) solves the AC problem by performing a *voting phase* and a *decision phase*. In the *voting phase*, the coordinator sends a *request-for-vote* message (also called a *prepare* message) to all the participants in the transaction. Each participant replies by sending its vote. If a participant votes *yes*, it enters a *prepared state* during which it can neither commit nor abort the transaction unless it receives the final decision from the coordinator. If, on the other hand, a participant votes *no*, it can unilaterally abort its transaction branch. The period of time from the moment a participant votes *yes* and until it receives the final decision is called the *uncertainty period* for that participant. During the *decision phase*, the coordinator decides on the transaction and sends its decision to all the participants. The coordinator's decision is *commit* if all participants have voted *yes*. Otherwise, the decision is *abort*. When a participant receives the final decision, it complies with this decision and sends back an acknowledgment, which is a promise from the participant that it will never ask the coordinator about the outcome of the transaction. This describes 2PC assuming no crash failures occur during the protocol execution.

3.2.2. Recovering from failures. Since we consider a *crash-recovery* failure model, participants can be down and later become operational again. In this case, the following *AC-Termination* property has to be added to the specification of the AC problem in order to exclude uninteresting protocols that allow participants to remain undecided forever once some crash failure has occurred during the protocol execution.

AC-Termination: If all failures are repaired, then unless a new failure occurs, every participant eventually reaches a decision.

Recovery is made possible by recording the progress of the protocol during normal processing (i.e., in the absence of failures) in the logs of the coordinator and the participants. Since failures can occur at any time, some of the information stored in the logs must be *force-written* (i.e., written immediately to a stable storage that sustains failures). For instance, the coordinator force-writes its decision before sending it to the different participants. Each participant force-writes (1) its vote before sending it to the coordinator, and (2) the final decision before acknowledging the coordinator. Usually, a participant that votes *yes*

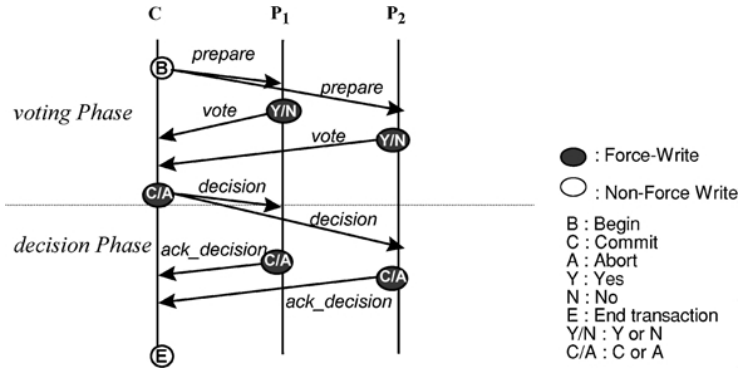

Figure 1. The Basic 2PC protocol.

force-writes its vote together with all the updates performed on behalf of the transaction. This ensures that the participant's updates are permanent even if it crashes (i.e., to ensure transaction *resiliency*). Force-writing a decision record in the log is the act by which a process *decides* on the transaction. Figure 1 describes the protocol execution between a coordinator C, and two participants P_1 and P_2, where both the C and P_1 roles can be performed by the same physical site.

To satisfy the *AC-Termination* property, specific actions that deal with crash failures must be supplied. Consider first a coordinator crash that occurs during the protocol execution. Assuming that timeouts are used to detect crash failures, if a participant P_i times out waiting for the *prepare* message, it can unilaterally decide *abort*. If, on the other hand, P_i times out waiting for the decision message (i.e., while in its uncertainty period), it cannot decide on its own. In this case, P_i starts a *termination protocol* during which it tries to find out what to decide by contacting another participant that either (i) knows the decision, or (ii) can unilaterally decide on the transaction. If however, due to crash failures, all participants satisfying (i) or (ii) are down, P_i remains blocked until at least one such participant recovers from its crash. When used with 2PC, this termination protocol satisfies the *AC-Termination* property. Indeed, if all participants that are down eventually recover and remain operational long enough, P_i will be able to communicate with at least one participant that satisfies (i) or (ii), namely the coordinator.

Consider now a participant P_i recovering from a crash. A failed participant returns to the operational state by executing a *recovery protocol*. During this protocol, P_i first restores a consistent local state using the information it stored in its stable log. Then, it tries to decide on the transactions that were active at the time the crash occurred (i.e., transactions for which no decision record exists in the log). For each of these transactions, if P_i does not find a *yes* record in its log, it can unilaterally decide *abort*. If, on the other hand, a *yes* record is found, this means that P_i failed while in its uncertainty period, and therefore, P_i is exactly in the same state as if it had timed out waiting for the decision message. Thus, the termination protocol described above can be used to decide on the transaction.

Finally, note that since the coordinator of a transaction has no uncertainty period, it can always decide on the transaction. Indeed, if the coordinator times out waiting for a participant's vote, it can safely abort the transaction. Similarly, while recovering from a crash, the coordinator can abort all transactions that were active at the time its crash occurred.

3.3. Optimized 2PC variations

As stated in the introduction, 2PC introduces a substantial delay in the system even in the absence of failures. Assuming that n is the total number of participants, 2PC requires 3 communication steps (*request-for-vote*, *vote* and *decision*) and $2n + 1$ forced log writes until a decision is reached at every participant. The number of force-writes performed is very important since it determines the number of blocking I/O required for a good behavior of the protocol. Furthermore, 2PC has a high message complexity due to $4n$ messages (including the acknowledgement of the decision) exchanged during the protocol execution.

The basic 2PC requires information to be explicitly exchanged and logged whether the transaction is to be committed or aborted. Several 2PC optimizations that make some presumptions about missing information were proposed:

The *Presumed Abort* optimization (PrA) [17] was designed to reduce the cost associated with aborting transactions. The coordinator of the protocol does not log information nor wait for acknowledgments regarding aborted transactions. Consequently, participants do not acknowledge abort decisions nor log information about such decisions. In the absence of information about a transaction, the coordinator presumes that the transaction has been aborted.

The *Presumed Commit* protocol (PrC) [17] is the counterpart of PrA in the sense that it reduces the cost associated with committing transactions. In PrC, the coordinator interprets missing information about transactions as *commit* decisions. Unlike PrA, however, the coordinator of PrC has to force-write a *membership* log record that contains the identities of all the participants in the transaction, and that, before starting the voting phase of the protocol. This is to ensure that an undecided transaction is not presumed as committed when the coordinator recovers from a crash.

The latency of an ACP is determined by the number of forced log writes and communication steps performed during the execution of the protocol, and until a decision is reached at every participant.[5] Table 1 shows the performances of 2PC together with its optimizations

Table 1. The cost of transaction commit.

	Message complexity	Latency	
		Forced log writes	Communication steps
2PC	$4n$	$2n + 1$	3
PrA	$4n$	$2n + 1$	3
PrC	$3n$	$n + 2$	3

in terms of latency and message complexity needed in order to commit a transaction. When compared to 2PC, PrA does not reduce the cost of committing transactions. PrC requires fewer forced log writes and messages than 2PC but does not reduce the number of communication steps required to commit a transaction.[6]

Beside this inefficiency, and as already mentioned in the introduction, 2PC and its variations violate site autonomy, consume valuable resources, and increase the probability of aborting a successfully processed transaction. We believe that these limitations, which are particularly exacerbated in today's distributed systems and applications, constitute a strong argument towards a serious reconsideration of two-phase commit, and explain the renewed interest in the atomic commitment problem.

4. Dictatorial atomic commitment

4.1. Informal description

Variations of 2PC solve the classical *Atomic Commitment* problem (specified in [5]) by performing a *voting phase* and a *decision phase*. The possibility of a participant to vote *no* reflects its ability to reject a transaction a posteriori, i.e., after the transaction's operations are processed. In particular, a participant might need to vote *no* if it detects a risk of violating any of the local ACID properties of its transaction branch. Obviously, if we remove the *veto right* from participants in atomic commitment, the coordinator will not need to ask the participants for their votes and the *voting phase* of a 2PC becomes useless (cf. figure 1).

Based on this idea, several authors have proposed the use of *One-Phase Commit* (1PC) protocols [3, 4, 21, 22]. The basic assumption underlying 1PC is that a participant "does not need" to vote. This actually means that before triggering the commit protocol, the coordinator of a 1PC makes sure that the ACID properties of the local transaction branches are already ensured at all participants. In other words, the coordinator of a 1PC acts as a *nice dictator* and makes sure that no participant can have any reasonable reason to vote *no*. Obviously, this introduces some assumptions on the way participants manage their transactions as will be detailed later in the section.

4.2. The dictatorial atomic commitment problem

The problem solved by 1PC is not the classical *Atomic Commitment* problem (as specified in [5]) anymore. This would contradict well-known lower bounds on the cost of solving atomic commitment in distributed systems [9].

In this problem participants do not have the *veto right*. At commit time, the coordinator proposes one of two values: *commit* or *abort*. If the coordinator does not crash, it forces the participants to accept its proposed value so that either they all commit the transaction or they all abort it. We formalize these notions as a set of properties that define the underlying problem, which we call *Dictatorial Atomic Commitment* (DAC).

DAC-Agreement: No two participants reach different decisions.
DAC-Validity: If the coordinator does not crash, the decision value is the coordinator's proposed value.
DAC-Integrity: No participant can reverse its decision after it has reached one.

The proposed value of the coordinator depends on whether or not the transaction has been successfully processed. A transaction is considered as successfully processed if all of its operations have been successfully executed and acknowledged by all participants. In this case, the coordinator proposes *commit*; otherwise, it proposes *abort*.

4.3. Basic one-phase commit (1PC)

All 1PC protocols that were proposed in the literature share the same basic structure and differ only in the way recovery is managed. This section describes the basic 1PC protocol around which all variations were designed and proves its correctness.

4.3.1. Protocol description. The simplest way to solve the DAC problem defined above is through the *terminate( )* function described in figure 2.

During this function, the coordinator decides on the transaction depending on its *proposition* value, and sends its *decision* to all participants in the transaction. When a participant receives the *decision* from the coordinator, it decides on the transaction. Note that force-writing a *decision* record in the log is the act by which a participant decides on a transaction. The protocol corresponds exactly to a 2PC without the voting phase (see figure 1). Clearly, one can apply various well-known optimizations of 2PC (e.g., *Presumed Commit*, *Presumed Abort*) to 1PC.

4.3.2. Protocol correctness. In this section, we prove the correctness of the basic 1PC protocol presented in figure 2 by showing that it satisfies all of the three properties of the DAC problem.

```
function terminate ()

Only the coordinator executes:
1  decision: = proposition;                    // proposition ∈ {commit, abort}
2  decide (decision);
3  send (decision) to all other participants;
4  return;

Every participant P_i ≠ coordinator executes:
5  wait until [received (decision) from coordinator]
6        decide (decision);
7        return;
```

Figure 2. The Basic 1PC protocol.

Theorem 4.1. *1PC achieves the DAC-Agreement property.*

Proof (Sketch): For contradiction, assume that a participant P_i decides *commit*, while another participant P_k decides *abort*. In 1PC, a participant can only decide at line 6 following the receipt of the decision message from the coordinator (line 5). This means that the coordinator has sent two different decisions to participants P_i and P_k. This contradicts the fact that the coordinator sends the decision only once at line 3 of the protocol. Furthermore, it is clear that the decision sent by the coordinator at line 3 is nothing but the value it has decided at line 2. Thus, all participants (including the coordinator) reach the same decision. □

Theorem 4.2. *1PC achieves the DAC-Validity property.*

Proof (Sketch): From lines 1 and 2 of the protocol, it is obvious that the coordinator's decision value is its proposed value. By the *AC-Agreement* property, the decision value of all participants is the coordinator's proposed value. □

Theorem 4.3. *1PC achieves the DAC-Integrity property.*

Proof (Sketch): From the structure of the protocol, it is obvious that the coordinator decides at most once by executing line 2, while the other participants decide at most once by executing line 6. □

4.3.3. Assumptions on the transactional systems. By interpreting acknowledgement messages as *yes* votes, the coordinator of 1PC verifies whether or not the ACID properties of the local transaction branches are already ensured at commit time. This obviously introduces some assumptions on the way participants manage their transactions. More precisely:

1. 1PC assumes that every transaction operation is acknowledged. Consequently, if the coordinator receives the acknowledgement messages for all the transaction operations before the protocol is launched, the *Atomicity* of all the local transaction branches (i.e., local atomicity) will be already ensured at commit time.
2. 1PC assumes that integrity constraints are checked after each update operation and before acknowledging the operation. Thus, if all operations are acknowledged, *Consistency* of all the local transaction branches will be already ensured at commit time (e.g., the possibility of discovering, at commit time, that there is not enough money for a bank account withdrawal is excluded).
3. 1PC assumes that a transaction that executes successfully all of its operations can no longer be aborted due to a serialization problem. Consequently, if all operations are acknowledged, *serializability* (Isolation) of all the local branches will be already ensured at commit time (e.g., concurrency control protocols that check serializability at commit time are excluded).
4. Finally, 1PC assumes that once all operations are acknowledged, and before the protocol is launched, the effects of all the local transaction branches are already logged on stable storage, and hence, the *Durability* property will be ensured at commit time.

We believe that assuming every operation to be acknowledged before the ACP is launched is not a strong requirement as most transactional standards like DTP from X/Open [24] and OTS from OMG [18] assume the same behavior. Similarly, the assumption that integrity constraints are checked after each update operation is not constraining since it covers a wide range of applications. However, the consequences of the last two assumptions are clearly less obvious. In the following two sections, we dissect these two assumptions and study their impact on the concurrency control and recovery protocols employed by participants in dictatorial atomic commitment.

5. The impact of dictatorship on concurrency control

In this section, we characterize schedulers that are correct without the need for a *veto right* at commit time. We give two necessary and sufficient correctness properties of such schedulers. The first property is an extension of *serializability*, which we named *on-line serializability*, and the second is the well-known *cascadelessness* property [5]. We show for instance that either strict *Two-Phase Locking* or strict *Timestamp Ordering* is sufficient to ensure *on-line serializability* and *cascadelessness*.

5.1. Veto right free schedulers

The correctness of a scheduler is usually captured through two properties: *serializability* and *recoverability* [5]. That is, a scheduler S is *correct* if only histories that are *serializable* and *recoverable* are acceptable for S. Roughly speaking, a scheduler does not need a *veto right* if it does not rely on a distributed voting phase to ensure either of these properties. For instance, the scheduler cannot optimistically authorize conflicts and decide to abort transactions at their termination time if the conflicts persist. In other words, an optimistic certifier does need a *veto right*. To capture these intuitive ideas, we first define the notion of *committed extension* of a history.

Definition 5.1. Let H be any history. A *committed extension* of H is any history obtained by extending H with the commit operations of all active transactions in H.

Consider for example the following history:[7]

$$H = W_1[x]R_1[y]W_2[z]A_1W_3[x]R_3[x]W_4[z]C_2$$

Both histories $H1$ and $H2$ below are *committed extensions* of H.

$$H1 = W_1[x]R_1[y]W_2[z]A_1W_3[x]R_3[x]W_4[z]C_2C_3C_4$$
$$H2 = W_1[x]R_1[y]W2[z]A_1W_3[x]R_3[x]W_4[z]C_2C_4C_3$$

The following definition expresses the fact that a scheduler making use of 1PC (i.e., with *no veto right* at commit time) does not control the commitment of a transaction after its operations have been performed.

Definition 5.2. A scheduler S is *commit-expanded* if, whenever a history H is acceptable for S, any committed extension of H is also acceptable for S.

It is easy to see that a scheduler might be correct but not *commit expanded*. Let S be any correct scheduler (e.g., an optimistic certifier) for which the following history is acceptable:

$$H = W_1[x]R_2[y]W_2[x]R_1[x]$$

Now consider the following committed-extension of H:

$$H' = W_1[x]R_2[y]W_2[x]R_1[x]C_1C_2$$

The serialization graph of H' contains the cycle $T_1 \rightarrow T_2 \rightarrow T_1$, which means that H' is not *serializable*. The history H' is not recoverable either because transaction T_1 reads x from transaction T_2 and yet T_1 commits before $T_2(C_1 < C_2)$. As a consequence, H is acceptable for S whereas H' is not. In other words, S is not *commit-expanded*.

Definition 5.3. We say that a scheduler is *VR-free* (*veto right free*) if it is *correct* and *commit-expanded*.

5.2. *On-line serializability and cascadelessness*

The example above shows that *serializability* and *recoverability* are not sufficient for *VR-freedom*. In the following, we introduce a property, that we call *on-line serializability*, which is stronger than *serializability*. Then we show that *on-line serializability* and *cascadelessness* (a history H is *cascadeless* if no transaction in H reads from values written by uncommitted transactions) [5], are necessary and sufficient for *VR-freedom*.

To define *on-line serializability*, we introduce the *notation E-SG*(H) (*Expanded Serialization Graph*). Given a history H over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$, *E-SG*$(H)$ denotes the directed graph whose nodes are the transactions in T that are either committed or active in H and whose edges are all $T_i \rightarrow T_j (i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. Note that *E-SG*(H) is a super-graph of *SG*(H) (the serialization graph of H) as the latest contains only committed transactions of H.

Definition 5.4. We say that a history H is *on-line serializable* iff *E-SG*(H) is acyclic.

Theorem 5.1. *Let S be any* commit-expanded *scheduler. S is correct iff S ensures* on-line serializability *and* cascadelessness.

Proof (Sketch):

(if) Let S be any *commit-expanded* scheduler and assume that every history that is acceptable for S is *on-line serializable* and *cascadeless*. As for any history H, *E-SG*(H) is a super-graph of *SG*(H), any cycle in *SG*(H) appears in *E-SG*(H) as well. Hence, any history

that is not *serializable* is not *on-line serializable*. Furthermore, it was shown in [5] that any history that is *cascadeless* is *recoverable*. Hence S is correct.

(only if) We show now that if a *commit-expanded* scheduler does not ensure either *on-line serializability* or *cascadelessness*, then it cannot be correct. Assume by contradiction that there is a history H in S that is either (1) not *on-line serializable* or (2) not *cascadeless*. Case (1) means that there is a cycle in *E-SG*(H). Let H' be any *committed-extension* of H. As S is *commit-expanded*, then H' is acceptable for S. As *E-SG*(H) = *SG*(H'), then *SG*(H') also contains a cycle, a contradiction with the assumption that S is correct, i.e., S ensures *serializability*. Case (2) means that in H some transaction T_1 reads from values written by an uncommitted transaction T_2. Let H' be any *committed-extension* of H where T_1 commits before T_2. As S is *commit-expanded*, then H' is acceptable for S. Since H' contains all *reads* and *writes* operations of H, then in H', T_1 reads from values written by T_2, and T_1 commits before T_2 in H'. A contradiction with the fact that S is correct, i.e., S ensures *recoverability*. □

Corollary 5.1. On-line serializability *and* cascadelessness *are necessary and sufficient conditions for a scheduler to be* VR-free.

5.3. Examples of VR-free schedulers

We show below that a scheduler based either on strict *Two-Phase Locking* (2PL) or on strict *Timestamp Ordering* (TO) is *VR-free*.

Theorem 5.2. *Strict 2PL is sufficient but not necessary to ensure* on-line serializability *and* cascadelessness.

Proof (Sketch):

(a) It has been shown in [5] that any strict history is *cascadeless*. Assume H is also a 2PL history and assume by contradiction that H is not *on-line serialisable*, i.e., there is a cycle $T_1 \to T_2 \to \cdots \to T_n \to T_1$ in *E-SG*(H). However, since 2PL is a lock-based scheduler, a dependency cycle would have led to a deadlock, and H could not have been generated: a contradiction.

(b) The following history H shows that strict 2PL is not necessary to ensure on-line serialisability and cascadelessness:

$$H = W_1[x]W_2[x]C_2C_1$$

The history H cannot be generated by a 2PL scheduler: transaction T_2 could not have accessed x before the termination of T_1. However, H is *on-line serializable* and *cascadeless*. □

Theorem 5.3. *Strict TO is sufficient but not necessary to ensure* on-line serializability *and* cascadelessness.

Proof (Sketch):

(a) Similar to (a) of Theorem 5.2 above: assuming H is a TO history, the presence of a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ in *E-SG*(H) would mean that $\text{ts}(T_1) < \text{ts}(T_1)$, where $\text{ts}(T)$ denotes T's timestamp. A contradiction.

(b) The following simple history H shows that strict TO is not necessary to ensure *on-line serializability* and *cascadelessness*

$$H = W_1[x]W_2[x]C_1C_2$$

Whatever the timestamp order is, H cannot be generated by a strict TO scheduler. Indeed, either $\text{ts}(T_1) < \text{ts}(T_2)$ and $W_2[x]$ will be delayed until C_1 is performed, or $\text{ts}(T_2) < \text{ts}(T_1)$ and T_2 will be aborted because it arrives late. However, H is *on-line serialisable* and *cascadeless*. □

In contrast, a certifier cannot ensure *on-line serializability*. A certifier typically prevents cycles by aborting transactions (a posteriori). However, *on-line serializability* requires that no cycle (even if involving only active transactions) be ever generated. The following history can be produced by a certifier and is obviously not *on-line serializable*.

$$H = R_1[x]W_2[x]W_2[y]W_1[y]$$

5.4. Practical considerations

Strict 2PL is the most widely used serialization protocol. Hence, participants of most transactional systems exhibit the *VR-free* property and thus, are 1PC compliant. However, commercial database systems are likely to use *isolation levels* standardized by SQL2 [14] in combination with 2PL. We recall below the SQL2 isolation levels and analyze the extent to which 1PC protocols can accommodate them.

- *Serializable*: Transactions running at this level are fully isolated.
- *Repeatable read*: Transactions running at this level are no longer protected against phantoms. More precisely, successive reads of the same object give always the same result but successive SQL queries selecting a group of objects may give different results if concurrent insertions occur.
- *Read committed*: Transactions running at this level read only committed data but *Repeatable Read* is no longer guaranteed. In a lock-based protocol, this means that read locks are relaxed before transaction end (in practice, as soon as they are granted).
- *Read uncommitted*: Transactions running at this level may do dirty reads. For this reason, they are not allowed to update the database. In a lock-based protocol, this means that *Read Uncommitted* transactions do not request locks at all.

Isolation levels are widely exploited because they allow faster executions, increase transaction parallelism and reduce the risk of deadlocks. For example, a transaction T_i computing

statistics on a large population of objects can take benefit of the *Read Uncommitted* level. This transaction will never be blocked by concurrent writing transactions (that may affect T_i's result but in a non significant way) and will never block other transactions.

If we refer to Definition 5.3, it is clear that schedulers implementing isolation levels, which we call *IL-schedulers*, are not *VR-free* simply because they are not correct: they do not ensure *serializability*. Consequently, they do not ensure *on-line serializability* either. However, isolation levels have been actually introduced to relax serializability, and non-serializable schedules that may be produced are considered as semantically correct. Hence, new correctness criteria that accommodate isolation levels need to be defined in order to characterize "correct" *IL-schedulers*. To this end, we introduce in the following a new property, which we call *IL-serializability*.

Consider a history H over a set of transactions $T = \{T_1, T_2, \ldots, T_n)$. Let *IL-SG*($H$) be the sub-graph of *SG*(H) containing all dependencies in H except those incurred by conflicts ignored by the isolation levels under which transactions in T are running. We say that H is *IL-serializable* iff *IL-SG*(H) is acyclic. An *IL-scheduler* is said to be *correct* if it ensures *IL-serializability* and *recoverability*.

Similarly to Section 5.2, we introduce *on-line IL-serializability* to characterize *IL-schedulers* that are correct with no *veto right* at commit time. Let *E-IL-SG*(H) denote the expanded *IL-SG*(H). We say that a history H is *on-line IL-serializable* iff *E-IL-SG*(H) is acyclic. We can show that *on-line IL-serializability* and *cascadelessness* are necessary and sufficient conditions for an *IL-scheduler* to be *veto right free*. The proof is very similar to that of Theorem 5.1 and hence omitted.

We show below that *IL-2PL* (2PL based *IL-scheduler*) satisfies both *cascadelessness* and *on-line IL-serializability*.

- *Cascadelessness*: conventionally, the *cascadelessness* property precludes the occurrence of dirty reads. In *IL-2PL*, dirty reads are allowed only at the *Read Uncommitted* level, which is restricted to *Read-Only* transactions. However, the semantics of *Read-Only* transactions contradict the fact that they can be subject to cascading aborts. Consequently, *cascadelessness* is still ensured in *IL-2PL* schedulers.
- *On-line IL-serialisability*: Assume H is an *IL-2PL* history, and assume by contradiction that H is not *on-line IL-serialisable*, i.e., there is a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ in *E-IL-SG*(H). Note that any dependency edge in *E-IL-SG*(H) translates a conflict not ignored by the *IL-2PL* scheduler. Since *IL-2PL* is based on locking, a dependency cycle would have led to a deadlock and H could not have been generated: a contradiction.

As a conclusion, *IL-2PL* schedulers can still be considered as *veto right free*, and hence they comply with 1PC.

6. The impact of dictatorship on recovery

A data manager must ensure the *Atomicity* and *Durability* properties of every transaction. More precisely, the data manager must guarantee that there is enough information on stable storage so that if a failure occurs (and the information in the volatile storage is lost),

(1) the updates of aborted transactions are undone from the database and (2) the updates of committed transactions are correctly reported on the database. Following the terminology of [13], we call the first property *abort-resiliency* and the second property *commit-resiliency* (these correspond to *undo* and *redo* rules respectively in [5]). A data manager is said to be *correct* if it guarantees both *abort-resiliency* and *commit-resiliency* [13].

6.1. Veto right free data managers

In a centralized system, *abort-resiliency* is for example ensured by having the data manager store *before images* in its log (this technique relies on the assumption that a strict concurrency control is used), and *commit-resiliency* is ensured by force-writing the transaction updates on stable storage at commit time [5].

In a distributed database system, the same technique is used to guarantee *abort-resiliency*. To ensure *commit-resiliency*, participants in a transaction must guarantee that, if the transaction commits at any participant, there is enough information on stable storage to *redo* the effects of the transaction at *all* participants. With a 2PC, this is guaranteed using the notion of *prepared* state. A participant P enters the *prepared* state for a transaction only if the *commit-resiliency* property is guaranteed for the transaction branch that accessed P. To commit a transaction, its coordinator makes sure that all updated participants have entered the *prepared* state of that transaction: this test is included in the *voting* phase of the 2PC. A participant does only vote *yes* if it has entered the *prepared* state. If it cannot enter that state (e.g., if the disk is full), the participant simply votes *no* and the transaction is aborted.

Removing the *veto right* has no impact on *abort-resiliency*. Nevertheless, the participants must anticipate the commit and make sure *the commit-resiliency* property is ensured a priori. As for schedulers, we introduce the following definitions to capture the idea of a *VR-free* data manager.

Definition 6.1. We say that a data manager D is *commit-expanded* if whenever an operation has been performed on behalf of a transaction T, the corresponding transaction branch can commit.

The definition above captures the idea that (just like for a scheduler), the only way to abort a transaction is by not performing one of its operations. If a transaction's operation has been acknowledged (i.e., performed), the corresponding transaction branch is able to commit.

Definition 6.2. We say that a data manager is *VR-free* if it is *correct* and *commit-expanded*.

6.2. On-line commit-resiliency

We introduce the following property to characterize the behavior of data managers that are *VR-free*.

Definition 6.3. We say that a data manager ensures *on-line commit-resiliency* if every *update* operation executed on that data manager is *commit-resilient*.

Theorem 6.1. *Let D be any* commit-expanded *data manager. D is* correct iff *it ensures* abort-resiliency *and* on-line commit-resiliency.

Proof (Sketch):

(if) Let D be any *commit-expanded* data manager that ensures *abort-resiliency* and *on-line commit-resiliency*. In other words, before acknowledging any update operation, the participant force-writes its effects on stable storage. As we assume that this participant cannot commit its transaction branch before all of its operations have been acknowledged (cf. Section 4), this means that it cannot commit its transaction branch if the effects of any of its operations are not on stable storage, i.e., the transaction is *commit-resilient* at D's site. Hence, D is correct.

(only if) Assume by contradiction that there is an execution where D does not ensure *on-line commit-resiliency*, i.e., D does not ensure the *commit-resiliency* of some update operation *op* for a transaction T. If the transaction commits exactly after receiving the acknowledgement from the participant about the operation *op*, and the participant crashes immediately after sending back that acknowledgment, then the effects of *op* are lost and T is not *commit-resilient* at D's site: a contradiction with the fact that D is correct. □

Corollary 6.1. Abort-resiliency *and* on-line commit-resiliency *are necessary and sufficient conditions for a data manager to be* VR-free.

6.3. Practical considerations

6.3.1. Participant logging. To achieve the *on-line commit-resiliency* property, participants in a transaction must force-write the effects of every update operation on stable storage, and that before acknowledging the operation. The *Early Prepare* (EP) processing scheme of Stamos and Cristian does ensure that property [21, 22]. Although *Early Prepare* can make use of 1PC and alleviates the need for an expensive 2PC, it requires a forced-write at every update operation of the transaction. The cost of transaction commitment is hence traded with the cost of transaction processing.

6.3.2. Coordinator physical logging. To avoid the prohibitive cost of *on-line commit-resiliency*, one might deviate from the "classical" atomic commitment scheme that requires every participant to ensure all of the ACID properties of its transaction branches. Consider for instance a less classical scheme that consists in having the coordinator itself ensure the *commit-resiliency* property before committing a transaction. To delegate this responsibility, the participants need however to make sure that the coordinator has enough information on its local stable storage about all committed transactions (unless it has the adequate information, the coordinator aborts the transaction). *Coordinator Log* (CL) [21, 22] and *Implicit Yes-Vote* (IYV) [3, 4] do follow this scheme.

In *Coordinator Log*, participants do not maintain their updates in a local stable log. Instead, they send back within the acknowledgment message of every update operation all the log records (undo and redo log records) generated during the execution of the operation. The coordinator is thus in charge of logging the transaction updates before performing the commit protocol. If we refer to the basic 1PC protocol described in Section 4, this would mean that the coordinator of CL calls the *terminate*( ) function with *commit* as its proposition value only if it succeeds in storing the transaction updates on stable storage. To recover from a crash, a participant asks the coordinator for the log records it needs to reestablish a consistent state of its database.

The *Implicit Yes-Vote* scheme is similar, except that logging is a more distributed task. The idea is to allow failed participants to perform part of the recovery procedure (the undo phase) independently of the coordinator, and to resume the execution of transactions that are still active in the system (i.e., transactions for which no decision was made yet) instead of aborting them. Participants send back their redo log records together with a *Log Sequence Number* (LSN) [12] whenever they acknowledge an update operation. To recover from a crash, a participant performs the undo phase of the recovery procedure and part of the redo phase using its local log. Then, the participant asks the coordinator for all redo log records whose LSNs are greater than its own highest LSN, and for all read locks acquired by active transactions. This allows the participant to reinstall the updates pertaining to the globally committed transaction and continue the execution of transactions that are still active in the system.

6.3.3. Coordinator logical logging. Although *Coordinator Log* and *Implicit Yes-Vote* circumvent the need for *on-line commit-resiliency*, they violate site autonomy by forcing participants to externalize their redo logs. This compromises their use in existing transactional systems. To solve this problem, we propose to maintain in the log of the coordinator the list of operations submitted to each participant instead of the physical redo log records sent back by these participants. In case a participant crashes during the 1PC protocol, the failed transaction branches that make part of a globally committed transaction will be re-executed using the operations registered in the coordinator's log. This mechanism, which we call *Coordinator Logical Log* (CLL), provides three main advantages. First, it preserves site autonomy since no internal information has to be externalized by the participants. Second, it can be applied to heterogeneous transactional systems using different local recovery schemes. Finally, it does not increase the communication cost during normal processing (redo log records are not piggybacked in the messages).

6.3.3.1. CLL: Description. As introduced before, our logical logging mechanism consists in having the coordinator register in its log every transaction operation before sending it to the participant hosting the object involved by the operation. Note that this registration is done by a non-forced write. Non-forced writes are buffered in main memory and do not generate blocking I/O. Operations are then sent to and locally executed by the different participants.

As is the case in CL and IYV, the coordinator of CLL is in charge of ensuring the *commit-resiliency* property before committing a transaction. Thus, when all acknowledgments are received, the coordinator force-writes the transaction operations on stable storage and calls

the *terminate*( ) function with *commit* as its proposition value. Recall that during this function, the coordinator decides on the transaction by force-writing its decision value on disk. However, in order to improve performances, the transaction operations together with the decision log record can be forced on stable storage at the same time, thereby generating a single blocking I/O.

If, on the other hand, the coordinator receives a negative acknowledgement from some participant or fails in storing the transaction operations on stable storage, it simply discards all the transaction log records and calls the *terminate*( ) function by proposing *abort*.

6.3.3.2. CLL: Recovering from failures. Consider a participant P_k recovering from a crash. Figure 3 details the recovery algorithm associated with CLL and executed by P_k. In the following, T_{ik} denotes the local branch of transaction T_i executed at participant P_k. For the sake of clarity, step numbers correspond here to step ordering.

Step 1 and Step 2 represent the standard local recovery procedure executed by a crashed participant P_k. To preserve site autonomy, we make no assumptions whatsoever on the way these steps are handled. Step 3 is necessary to determine if the kth branch (i.e., T_{ik}) of some globally committed transactions T_i has to be locally re-executed by the crashed participant.

In Step 4, the coordinator aborts all active transactions in which P_k participates. Step 5 checks if there exists some committed transaction T_i for which P_k did not acknowledge the *commit* decision. This may happen in two situations. Either the participant crashed before the commit of T_{ik} was achieved and T_{ik} has been undone during Step 1, or T_{ik} is locally

Participant's algorithm

1- *undo the transactions that were active at the time the crash occurred*
2- *redo the transactions that have reached their commit state before the crash occurred*
3- *contact the coordinator*
7- *answer the queries that may be sent by the coordinator during step 6*
10- *accept new transactions*

Coordinator's algorithm

If contacted by participant P_k during step 3, do:
for each transaction T_i in which P_k participates
4- *if ($commit_i$) $\notin$ coordinator's log, then*
broadcast ($abort_i$) to all other T_i participants and forget T_i
5- *if ($commit_i$) $\in$ coordinator's log, then*
if $ack_k(commit_i) \notin$ coordinator's log, then
6- *query P_k to determine the exact status of T_{ik}*
(i.e., either locally committed or aborted)
8- *if T_{ik} has not been locally committed, then*
restart a new transaction T'_{ik} on P_k
re-execute all T_{ik} operations within T'_{ik}
9- *accept new distributed transactions*

Figure 3. Recovering a participant crash.

committed but the crash occurred before the acknowledgment was sent to the coordinator. Note that these two situations must be carefully differentiated. Re-executing T_{ik} in the latter case may lead to inconsistencies if T_{ik} contains *non-idempotent* operations.

To simplify the presentation, we assume for the moment that the coordinator can query a participant to learn the exact state of T_{ik} (Step 6). We detail afterwards the way we achieve this without violating site autonomy. The participant answers during Step 7. If T_{ik} has been successfully committed, the coordinator does nothing. Otherwise, T_{ik} has been undone during Step 1 and must be entirely re-executed. This re-execution is performed by exploiting the coordinator's log (Step 8). Once the recovery procedure is completed, new distributed transactions are accepted by the coordinator (Step 9) and the participant (Step 10).

We now explain how the coordinator can query a participant about the state of its local transaction branches. Our solution relies on a local *Agent* (called $Agent_k$) associated with each participant P_k. The *Agent* does not violate site autonomy as the existing interface of the participant is preserved, and does not increase the communication cost, as it is co-located with its participant. Every message is submitted to the participant through its local *Agent*, which acts as a liaison between the coordinator and the participant. The exact role of the *Agent* is to determine, during the recovery procedure, those local transaction branches that need to be re-executed. The mechanism works as follows. When the coordinator broadcasts the *commit* decision to each participant, the participant's *Agent* issues an additional operation "write record <commit$_i$>" on behalf of the local transaction branch it is in charge of (e.g., T_{ik}), and before submitting the *commit* decision to the participant.[8] This creates at P_k a special local record containing the *commit* decision for T_i. This operation will be treated by P_k in exactly the same manner as the other operations belonging to T_{ik}, that is, either all committed or all aborted atomically. Once the *Agent* receives the acknowledgment of this write operation, it asks P_k to commit the local transaction branch.

Steps 6 and 7 of the recovery algorithm are now straightforward. To get the status of a local transaction branch T_{ik}, the coordinator checks, through $Agent_k$, the existence of record <commit$_i$> at P_k (this can be done by a regular select operation). If the record is found, this proves that T_{ik} has been successfully committed at P_k before the crash, since write <commit$_i$> is performed on behalf of T_{ik}. Otherwise, T_{ik} has been backward recovered during Step 2 and must be re-executed.

6.3.3.3. CLL: Recovery correctness. Similarly to the AC problem, the following *DAC-Termination* property has to be added to the specification of the DAC problem in order to exclude protocols that allow participants to remain undecided forever once a crash failure has occurred during the protocol execution.

DAC-Termination: If all failures are repaired, then unless a new failure occurs, every participant eventually reaches a decision.

In this section, we show that the CLL's recovery procedure described in figure 3 is correct. This amounts to proving that a recovering participant eventually reaches a decision consistent with that reached by the other participants once all failures are repaired so that *DAC-Termination* is satisfied. However, since the recovery procedure may lead to a decision

through the re-execution of a transaction branch, we also need to show that re-executing the logical operations registered in the coordinator's log will produce exactly the same local state at the recovering participant as the one produced during the initial execution. In the following, we consider these two issues in turn.

- *Decision consistency*: Let P_k be the recovering participant. If, during its local recovery procedure, P_k finds in its log a decision record for a transaction branch, say T_{ik}, then it has already decided during the 1PC protocol execution. If, however, no decision record is found, P_k undoes the effects of T_{ik} (Step 1). Note that the only non-trivial case to consider here is the case where T_{ik} is part of a globally committed transaction T_i. This may happen if the coordinator has sent the commit decision to all participants, but P_k crashed before committing T_{ik}. By the algorithm of figure 3, when P_k establishes a consistent local state, it contacts the coordinator (Step 3). In this case, once the coordinator has verified, through $Agent_k$, that T_{ik} has been locally aborted, it re-executes all T_{ik} operations within a new transaction branch T'_{ik}. If a failure should occur during the re-execution process, it will be retried until T_{ik} (T'_{ik}) commits at P_k. Note that although P_k may be blocked during its recovery (in case the coordinator is down), P_k eventually reaches a consistent decision once the coordinator recovers from its crash. Hence, the recovery procedure associated with CLL satisfies the *DAC-Termination* property.
- *Determinism*: Here, we show that the re-execution of T_{ik} within T'_{ik} produces the same local state at P_k as the one produced during the initial execution. Note that in CL and IYV, the coordinator's log contains physical redo records, making the recovery algorithm rather straightforward. The redo records are re-installed at the failed participant during the recovery of a local transaction branch, thereby producing the same local state as the one produced during the initial execution. By exploiting logical logging rather than physical logging, CLL's recovery procedure must face two new problems:
  - *Operations may be non-idempotent*: an operation *op* is said to be *non-idempotent* if $(op(op(x)) \neq op(x))$. Non-idempotent operations must be executed exactly once in any failure situation.
  - *Operations may be non-commutative*: two operations *op1* and *op2* are said to be *non-commutative* if $(op1(op2(x)) \neq op2(op1(x)))$. Non-commutative operations must be executed at recovery time in the same order as during the initial execution.

Consider first the management of non-idempotent operations. Assume the coordinator has decided to commit T_i and has sent its decision to the participants. Assume also that P_k crashed immediately after. By the *undo* rule, if P_k crashed before committing T_{ik}, T_{ik} will be undone during Step 1 of the recovery algorithm and the record $<\text{commit}_\text{i}>$ will be discarded.[9] Otherwise (i.e., P_k crashed after the commit of T_{ik} was successfully performed), the *redo* rule guarantees the presence of the $<\text{commit}_\text{i}>$ record at P_k. These two situations are differentiated during Step 6 of the recovery algorithm. Step 8 forward recovers only transaction branches that have been locally aborted. This means that no transaction branch, and hence no operation (either idempotent or not) is executed twice.

Consider now non-commutative operations. If these operations belong to the same transaction, no problem can occur. Indeed, the recovery algorithm re-executes the operations of a

failed transaction branch following the order in which they were logged on the coordinator, i.e., in the order of their initial execution. The case where two or more local transaction branches (eg., T_{ik}, T_{jk}) have to be forward recovered is more tricky since most transactional systems execute transactions in parallel through several threads of control. Thus, even if the coordinator re-submits to P_k all operations that belong to different local transaction branches in the order of their initial execution, the result is non-deterministic. We demonstrate below that the local database state produced by the recovery algorithm is the same as the one produced during the initial execution. Let φ denote the set of all local transaction branches that have to be forward recovered by P_k during Step 8.

$$\varphi = \{T_{ik}/\text{commit}_\text{i} \in \text{coordinator's log} \wedge \text{ack}_\text{k}(\text{commit}_\text{i}) \notin \text{coordinator's log} \wedge <\text{commit}_\text{i}> \notin P_k\text{'s state}\}.$$

First, Step 2 of the recovery algorithm guarantees that all resources accessed by any $T_{ik} \in \varphi$ are restored to their initial state (i.e., the state before T_{ik} execution), according to the *Atomicity* property. Second, since Step 8 precedes Step 9 and Step 10, new transactions that may modify T_{ik} resources are executed only after the re-execution of T_{ik}. Consequently, at Step 8, all $T_{ik} \in \varphi$ are guaranteed to re-access the initial database state. The sole problem may come from the parallel re-execution of all $T_{ik} \in \varphi$ if these transactions themselves compete on the same resources.

Assume first that P_k uses a locking based *VR-free* serialization protocol, such as strict 2*PL* (i.e., the general case). In this case, $\forall T_{ik}, T_{jk} \in \varphi, \neg\exists(T_{jk} \rightarrow T_{ik})$, where $\rightarrow$ represents a precedence in the serialization order. Otherwise, T_{ik} would have been blocked during its initial execution, waiting for the termination of T_{jk}, and would not have completed all its operations, which contradicts $T_{ik} \in \varphi$. This means that T_{ik} and T_{jk} cannot compete on the same resources. If however, P_k uses another *VR-free* serialization protocol, such as strict TO, the former assumption is no longer valid. Indeed, strict TO accepts some Read/Write conflicts (those produced in the timestamp order) without blocking. To deal with this case, Step 8 must execute all $T_{ik} \in \varphi$ in their initial serialization order, one after the other (i.e., without parallelism).

6.3.3.4. CLL: Timeout actions. To complete our discussion on CLL, and to satisfy the *DAC-Termination* property, we must supply timeout actions so that participants do not wait indefinitely for messages that may never arrive due to a coordinator crash.

Remember that in 1PC, the only point where a participant can unilaterally abort a transaction is by negatively acknowledging an operation. If, however, the participant has no pending acknowledgement for any of the transaction operations, it enters its uncertainty period until it receives either a new transaction operation or the final decision from the coordinator.

When a participant times out while in its uncertainty period (due to a coordinator crash), it executes a termination protocol during which it tries to decide on the transaction. The termination protocol presented in Section 3.2.2 can be perfectly used here so that *DAC-Termination* is guaranteed. Note that although the participant may be blocked during the execution of the protocol due to the crash of other participants and/or the coordinator, it eventually reaches a consistent decision once these crash failures are repaired.

Table 2. The cost of transaction commit in different 1PC variations.

	Message complexity	Latency	
		Forced log writes	Communication steps
EP	n	$1+n+op$	1
CL	n	1	1
IYV	$2n$	$1+n$	1
CLL	$2n$	$1+n$	1

6.4. Evaluation of CLL

In this section, we evaluate CLL along with existing 1PC variations, namely, EP, CL and IYV. Table 2 shows the performance of the different protocols in terms of latency and message complexity needed in order to commit a transaction. We denote by n the number of participants and by *op* the number of update operations performed by a transaction.

From Table 2, IYV and CLL have exactly the same performances. EP and CL have the lowest message complexity, which comes, however, at a high additional cost. Indeed, CL eliminates the acknowledgement of (*commit* and *abort*) decisions at the cost of a coordinator's log that is never garbage collected! Since participants do not acknowledge decisions, they do not need to force-write these decisions in their logs. This is how CL has also the smallest number of log forces. If the coordinator's log in CL has to be garbage collected, CL would have exactly the same performances as IYV and CLL.

EP is the 1PC variation of PrC, and hence, commit decisions are neither acknowledged nor force- written by the participants. Also, and just like PrC, EP requires that a *membership* log record[10] be force-written in the coordinator's log. However, unlike PrC, the coordinator of EP has to update its membership record each time a new participant joins the transaction. This generates $1+n$ log forces, assuming that participants are not known in advance. Furthermore, by achieving *on-line commit resiliency* (cf. Section 6), EP generates one force-write for each update operation, which makes a total of $1+n+op$ log forces.

Finally, it is very important to note that among all atomic commitment protocols, CLL is the only protocol that has the advantages of 1PC while preserving site autonomy. Thus, it is the sole protocol that can be applicable to all existing commercial transactional systems (that may not be 2PC compliant) without requiring any modification to the kernel of these systems so that they can participate in the protocol execution.

7. Integrating 1PC into standard platforms

In this section, we show the way 1PC can be integrated into well-known transactional standards, namely OMG's *Object Transaction Service* (OTS) [18] based on a CORBA architecture [19]. Our choice is motivated by the fact that CORBA allows for heterogeneity, and complies with other well-known and widely supported transactional standards, namely the X/Open DTP model [24]. We first recall some background related to the CORBA object

model and its associated Object Transaction Service. We then study how to extend OMG's OTS to support the 1PC behavior.

7.1. Background

7.1.1. CORBA. An important feature of today's distributed systems is *heterogeneity*. Applications autonomously developed at different sites in different languages and on different hardware and software platforms need to communicate and to share information. The *Common Object Request Broker Architecture* (CORBA) is an open standard, specified by the *Object Management Group* (OMG) [19], which answers the need for interoperability between these applications. Simply stated, CORBA provides a distributed object-oriented infrastructure that allows objects to communicate across boundaries such as the network, the specific language in which they were written or the platform on which they are deployed.

7.1.2. OTS. The OMG's *Object Transaction Service* (OTS) [18] brings the notion of distributed transactions to the world of CORBA. OTS provides interfaces that allow multiple, distributed objects to cooperate in a transaction such that they either all commit or all abort their changes. These interfaces are defined in terms of the *OMG's Interface Definition Language* (IDL). Figure 4 illustrates the major components and interfaces defined by the Object Transaction Service.

The transaction originator is an arbitrary program that begins a transaction using a *TransactionFactory*. A *Control* Object is returned that provides access to a *Terminator* and a *Coordinator*. The transaction originator uses the *Terminator* to commit or rollback the

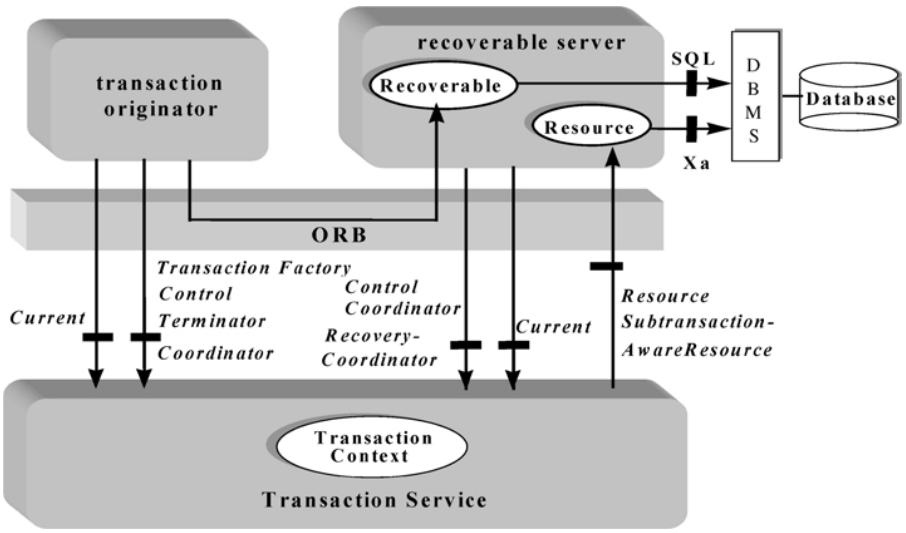

Figure 4. Object transaction service architecture.

transaction. The *Coordinator* provides mechanisms to coordinate the actions of the different participants involved in the transaction.

A recoverable server implements one or more recoverable objects. A recoverable object is an application object that manages persistent data whose state is subject to change during the course of the transaction, and thus must participate in the commit protocol driven by the Transaction Service.[11] This is achieved by registering a *Resource* object that implements the 2PC protocol with the *Coordinator*. In other words, the *Resource* object plays the role of a participant in the commit protocol.

Note that one of the major goals of the Object Transaction Service is to allow several legacy (possibly heterogeneous) TP based systems to participate in a CORBA transaction. In particular, OTS is designed to be compatible with X/Open DTP compliant data managers [24], known as X/Open *Resource Managers* or RMs, while preserving their *autonomy* in the sense that no changes should be made to the participating RMs in order to accommodate the OTS model. If we refer to figure 4, this means that if the persistent data are managed by an X/Open RM, the OTS *Resource* object translates every OTS invocation to 2PC participants into an X/Open Xa invocation.

To commit a transaction, the Transaction Service drives the commit protocol by issuing requests to all the *Resources* registered with the *Coordinator*. Finally, and to simplify application programming, the OTS model also defines the *Current* interface that provides transparent access to the Transaction Service. Simply stated, *Current* can be seen as a high level API that hides the location of the Transaction Service and the set of its defined interfaces.

7.2. Integrating 1PC into OTS

As already stated before, an important objective of the Object Transaction Service is the integration of legacy systems in CORBA transactions. Recall from Section 6 that *Coordinator Logical Log* (CLL) is the only 1PC variation that preserves site autonomy, which is a preliminary condition towards this objective, while being compatible with commercial transactional systems.[12] This makes CLL the only possible candidate to the integration of 1PC into the OTS model so that it supports, instead of 2PC, the 1PC behavior.

Actually, one can think of two different approaches to achieve this goal. The first one consists in using an existing OTS product as a black box, and to add 1PC as an independent entity on top of it. Thus, the 1PC entity will access OTS only through its standard interfaces. However, a deep study showed that this is technically impossible to realize due to some constraints related to the ORB. We believe that the proof of this impossibility is out of the scope of this paper (details can be found in [1]).

The second approach consists in integrating 1PC within the kernel of an existing OTS implementation. Based on this approach, we present in the following a way by which Coordinator Logical Log can be embedded within a fully OTS compliant Transaction Service, named *MAAO OTS*, developed by the *TRANSREP* project at *INRIA* (Institut National de Recherche en Informatique et Automatique) [16]. Our solution has been fully designed and is currently being implemented in C++ using Orbix 2.3 MT [15], a commercial CORBA implementation, in the context of *OpenDREAMS-II*, an ongoing ESPRIT project [1] which aims at providing a CORBA compliant platform for reliable industrial applications.

7.2.1. Interface extension. MAAO OTS extension to support 1PC necessitates very minor changes to the standard OTS interfaces as defined by OMG. Indeed, only the Terminator interface is extended so that it supports a new operation, called *commit_1PC*( ), dedicated to the 1PC protocol. Furthermore, this operation is completely transparent to application programmers as will be shown later. This is mainly due to the fact that our solution was designed with the issue of application portability in mind.

7.2.2. Client's view. A client of the extended MAAO OTS will always access the standard interfaces as defined by OMG. The transparency of our 1PC extension to the client application is achieved through a library, called *1PClib*, dedicated to 1PC specific mechanisms and to which the client application should be linked. *1PClib* automatically maps the client call to the *commit*( ) operation to a call to the *commit_1PC*( ) operation on the *Terminator* with the appropriate parameters. The call to *commit_1PC*( ) launches the 1PC protocol implemented by the MAAO OTS server and commits the transaction in a single phase.

7.2.3. Recoverable server's view. Recall from Section 6 that the concept of *Agent* has been associated with each database system participating in the CLL protocol. The role of the *Agent* is to determine the exact state (i.e., *committed* or *aborted*) of every transaction branch for which its local DBMS did not acknowledge the commit decision due to a failure. This is important in order to identify those branches that need to be locally re-executed.

In our model, we have integrated the *Agent* role within the *Resource* object. Obviously, this is the most natural and straightforward way to do since the *Resource* is the entity that acts as intermediary between the Transaction Service and the participating DBMS. Furthermore, no interface extensions are needed since we exploit the standard *commit_one_phase*( ) operation offered by the *Resource* interface[13] to implement the *Agent* and to commit the transaction in a single phase on the different participants.

7.2.4. Failures and recovery. Recall that to be able to guarantee the *commit-resiliency* of transactions, the coordinator of CLL must keep in its log the list of application requests invoked by a transaction. In addition, the coordinator must force its log on stable storage before sending the commit decision to the different participants. In case a participant crashes during the 1PC protocol execution, the coordinator re-executes the transaction branch on the failed participant.

In OTS, the difficulty in realizing this is due to the fact that a transaction sends its requests directly to recoverable objects. Thus, at commit time, the coordinator has no knowledge of the list of requests invoked by the transaction during its execution.

To deal with this problem, our solution consists in keeping the list of a transaction requests in a log maintained on the client side. This log, which is completely transparent to the client application, will be managed by *1PClib* via an object called *Replay*. The *Replay* interface defines operations that allow to (i) write the transaction requests on the log (*register_op*( ) operation), (ii) force the log on stable storage (*flush*( ) operation), and (iii) re-execute the requests of a transaction branch in case a participant crashes (*re_execute*( ) operation).

7.2.5. Detailed description. *1PClib* has been implemented using Orbix Per-process Filters.[14] Once a client application is linked to *1Pclib* and the Per-process filter is installed,

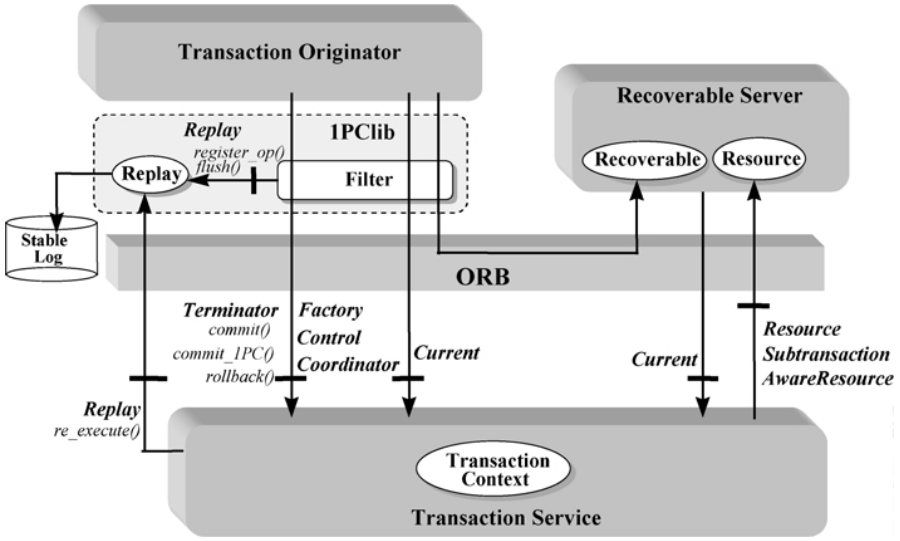

Figure 5. 1PClib in the extended MAAO OTS architecture.

the filter will monitor all out-going requests from the client's address space. Figure 5 describes the role of *1PClib* in the extended MAAO OTS to support the CLL protocol.

Typically, when a client application begins a transaction by calling *begin*( ) on the *Factory*, the *client filter* associates a *Replay* object with the new transaction. During the transaction, the client application invokes application requests on recoverable objects. The *client filter* intercepts each of these requests, registers the request in the log by calling *register_op*( ) on the *Replay* object, and continues the call normally. To commit its transaction, the client application calls *commit*( ) on the *Terminator*. Again, the client filter intercepts the call to *commit*( ), force-writes the transaction requests on stable storage by calling *flush*( ) on the *Replay* object, and maps the call to *commit*( ) by a call to *commit_1PC*( ) on the *Terminator*. In this call, the filter sends the *Replay* object reference associated with the transaction to the Transaction Service as a parameter of the operation *commit_1PC*( ), which launches the 1PC protocol in the MAAO OTS server and commits the transaction in a single phase.

In case a participant crashes during the 1PC protocol, the *Coordinator* of the transaction re-executes the failed branch by calling the operation *re_execute*( ) on the *Replay* object associated with the transaction.

8. Conclusion and future prospects

This paper discusses the impact of removing the *veto right* from the traditional *Atomic Commitment* (AC) problem, and gives a precise abstract specification of the resulting problem, which we call the *Dictatorial Atomic Commitment* (DAC) problem. One obvious impact is that a *One-Phase Commit* (1PC) protocol is sufficient to orchestrate transaction termination:

this saves two communication steps and associated forced-writes in comparison with the traditional 2PC protocol. A less obvious impact is a set of restrictions on concurrency control and recovery protocols employed by the participants in the transaction. We have captured those restrictions through three properties: *on-line serializability*, *cascadelessness* and *on-line commit-resiliency*. Those properties are strictly stronger than *serializability*, *recoverability* and *commit-resiliency*, respectively. Whereas *on-line serializability* and *cascadelessness* are realistic in most distributed database systems, *on-line commit-resiliency* can turn out to be very expensive.

To circumvent the need for *on-line commit-resiliency*, we discussed "non-classical" atomic commitment schemes that consist in having the coordinator ensure the *commit-resiliency* property before committing the transaction. We pointed out the limitations of existing protocols that are based on this scheme and proposed a new 1PC variation, named *Coordinator Logical Log* (CLL), which alleviates their drawbacks, making the 1PC idea indeed realistic and useful for most of today's distributed systems and applications. We also studied the implementation of our protocol in the context of current transactional standards, namely OMG's OTS.

Although the window of vulnerability to blocking in 1PC is larger than in 2PC (unless it has a pending acknowledgement for a transaction operation, a participant in 1PC enters its uncertainty period during which it cannot unilaterally decide on the transaction), non-blocking 1PC variations that give every correct participant the ability to eventually reach a decision despite the crash of the coordinator do exist. Indeed, in [2] we proposed a 1PC protocol, which, in addition to the properties of the DAC problem, ensures that every correct participant eventually reaches a decision.

Given the appealing features of CLL, we are currently investigating its adaptation to mobile and disconnected computing. A preliminary study showed that our protocol provides a suitable way of dealing with the problem of disconnections [7], and allows saving valuable and critical resources on data servers hosted by lightweight intelligent devices [6].

Finally, it would be very interesting to investigate intermediate cases between *veto rights for all* and *no veto right at all*. An interesting trade-off to better explore is then the relationship between the number of *veto rights* and the dependency of the recovery protocol (e.g., [4]).

Notes

1. Typical examples of SCS are command and control systems in the field of transport (air, railway and road) including traffic management and fleet management systems, technical management systems for large equipment infrastructures such as telecommunication networks or electricity and water distribution networks.
2. This integration has been realized in the context of the OpenDREAMS Esprit project. The main goal of the project is the design and implementation of a CORBA-compliant platform to support Supervision and Control Systems' applications.
3. Note that this does not exclude link failures, assuming that every link failure is eventually repaired.
4. This is generally the site where the transaction originated.
5. This definition of *latency* is based on the fact that the period of interest of an ACP is the time interval during which participants cannot relinquish valuable system resources they hold for exclusive use on behalf of the transaction.

6. This corresponds to the most frequent case since most transactions are expected to commit in the absence of failures.
7. In the notations, $R_i[x]$ and $W_i[x]$ denote respectively a Read (resp. Write) operation on object x performed by transaction T_i, while C_i and A_i denote the commit (resp. abort) of T_i.
8. Note that this operation never generates a dependency cycle (i.e., deadlock) since it is the last operation executed in any transaction that has to be committed.
9. We recall that the operation *write record* $<commit_i>$ is performed on behalf of T_{ik}.
10. The membership log record contains the list of participants in the transaction.
11. The commit protocol adopted by OMG for OTS is the standardized Presumed Abort (PrA) variation of basic 2PC.
12. No changes should be made to the kernel of existing commercial systems so that they can participate in the CLL protocol, which is not the case for IYV and CL protocols.
13. Note that although we exploit this function in order to implement the 1PC behavior, its semantics are totally different from the 1PC concept in the sense that it is basically supported by the *Resource* object in order to optimize the commit protocol in case of mono-site transactions.
14. Orbix filters have been normalized in CORBA 2.2 under the *Interceptors* concept.

References

1. M. Abdallah, C. Bobineau, R. Guerraoui, and P. Pucheral, "Specification of the transaction service," Esprit Project OpenDREAMS-II n°25262, Deliverable n°R13, 1998.
2. M. Abdallah and P. Pucheral, "A low-cost non-blocking atomic commitment protocol for asynchronous systems," in Proc. Int. Conf. on Parallel and Distributed Computing and Systems, 1999.
3. Y. Al-Houmaily and P.K. Chrysanthis, "Two-phase commit in gigabit-networked distributed databases," in Proc. Int. Conf. on Parallel and Distributed Computing Systems, 1995.
4. Y. Al-Houmaily and P.K. Chrysanthis, "The implicit-yes vote commit protocol with delegation of commitment," in Proc. Int. Conf. on Parallel and Distributed Computing Systems, 1996.
5. P.A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison Wesley: Reading, MA, 1987.
6. C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez, "PicoDBMS: Scaling down database techniques for the smartcard," in Proc. Int. Conf. on Very Large Data Bases, 2000.
7. C. Bobineau, P. Pucheral, and M. Abdallah, "A unilateral commit protocol for mobile and disconnected computing," in Proc. Int. Conf. on Parallel and Distributed Computing Systems, 2000.
8. Y. Breitbart, H. Garcia-Molina, and A. Silbershatz, "Overview of multidatabase transaction management," VLDB Journal, vol. 1, no. 2, 1992.
9. C. Dwork and D. Skeen, "The inherent cost of non-blocking commitment," in Proc. ACM Symposium on Principles of Distributed Computing, 1983.
10. J. Gray, "Notes on database operating systems," in Operating Systems: An Advanced Course, LNCS, vol. 60, Springer Verlag: Berlin, 1978.
11. J. Gray, "A comparison of the byzantine agreement problem and the transaction commit problem," in Fault-Tolerant Distributed Computing, LNCS, vol. 448, Springer Verlag: Berlin, 1987.
12. J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann: San Mateo, CA, 1993.
13. V. Hadzilacos, "A theory of reliability in database systems," Journal of the ACM, vol. 35, no. 1, 1988.
14. International Standardization Organization, Information Processing Systems-Database Language SQL, ISO/IEC 9075, 1992.
15. IONA, Orbix 2.3 Programmer's Guide, IONA Technologies Plc, 1997.
16. J. Liang, M. Saheb, and F. Giudice, "Maao OTS version2," ACTS Project ACTranS, Deliverable n°D2aa, 1998. Available at http://www.actrans.org/Publications.html.
17. C. Mohan, B. Lindsay, and R. Obermarck, "Transaction management in the R* distributed database management system," ACM Transactions on Database Systems, vol. 11, no. 4, 1986.
18. Object Management Group, Object Transaction Service, Document 97.12.17, OMG editor, 1997.

19. Object Management Group, The common object request broker: Architecture and specification, document 99.10.07, OMG editor, 1999.
20. A. Sheth and J. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," ACM computing surveys, vol. 22, no. 3, 1990.
21. J. Stamos and F. Cristian, "A low-cost atomic commit protocol," in Proc. IEEE Symposium on Reliable Distributed Systems, 1990.
22. J. Stamos and F. Cristian, "Coordinator log transaction execution protocol," Journal of Distributed and Parallel Databases, vol. 1, no. 4, 1993.
23. M. Stonebraker, "Concurrency control and consistency of multiple copies of data in distributed INGRES," IEEE Transactions on Software Engineering, vol. 5, no. 3, 1979.
24. X/Open CAE Specification, Distributed Transaction Processing: The XA Specification, XO/CAE/91/300, 1991.