One-Phase Commit : Does It Make Sense ?¹

Maha Abdallah*

* Laboratoire PRiSM
Université de Versailles
78035 Versailles, France
Firstname.Lastname@prism.uvsq.fr

Rachid Guerraoui**

Philippe Pucheral*
** Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
Rachid.Guerraoui@epfl.ch

Abstract

Although widely used in distributed transactional systems, the so-called Two-Phase Commit (2PC) protocol introduces a substantial delay in transaction processing, even in the absence of failures. This has led several researchers to look for alternative commit protocols that minimize the time cost associated with coordination messages and forced log writes in 2PC. In particular, variations of a One-Phase Commit (1PC) protocol have recently been proposed. Although efficient, 1PC is however rarely considered in practice because of the strong assumptions it requires from the distributed transactional system. The aim of this paper is to better identify and understand those assumptions. Through a careful look into the intrinsic characteristics of 1PC, we dissect the assumptions underlying it and we present simple techniques that minimize them. We believe that these techniques constitute a first step towards a serious reconsideration of 1PC in the transactional world.

1. Introduction

The transaction concept has proven to be very important for the design and implementation of fault-tolerant distributed applications [3]. A transaction is a set of operations updating shared objects and satisfying the so-called ACID properties, namely, Atomicity, Consistency, Isolation and Durability. The Atomicity property, also called all-or-nothing property, means that either the transaction successfully executes to completion and the effects of all of its operations are recorded in the

accessed objects (the transaction is said to be committed), or it fails and it has no effect at all (the transaction is aborted). Consistency means that the transaction does not violate the integrity constraints of accessed shared objects, while Isolation means that the intermediate effects of a transaction are not visible to concurrent transactions (Isolation has been formalized through the theory of serializability). Durability means that the updates of a committed transaction are permanent, e.g., stored on a stable storage that sustains failures.

To ensure the Atomicity property of a transaction accessing distributed data objects, all participants in the transaction must coordinate their actions so that they either unanimously abort or unanimously commit the transaction. This is achieved through an Atomic Commit Protocol (ACP), launched at the end of the transaction.

The best known ACP is the Two-Phase Commit protocol (2PC) [7], which has been adopted by almost all transactional standards, such as DTP from X/Open [5] and OTS from OMG [11], and implemented in several commercial database systems. Although widely used, 2PC is considered quite inefficient as it introduces a substantial delay in transaction processing. This delay is due to the time cost associated with the coordination messages and forced log writes.

These limitations have led many researchers to work on optimized versions or alternatives to 2PC. Among the optimizations of 2PC are the Presumed Commit (PrC), Presumed Abort (PrA) and Early Prepare (EP) protocols [10, 12]. These protocols reduce the cost of 2PC in terms of forced log writes and message complexity.

More recently, the idea of One-Phase Commit (1PC) suggested by Gray in [8] has been reconsidered, and variations of 1PC, such as Coordinator Log [12] and Implicit Yes-Vote (IYV) [1] protocols, have been proposed. 1PC further reduces the cost of atomic commit by eliminating the voting phase of 2PC: two

¹ This work has been partially funded by the CEC under the OpenDREAMS Esprit project n°20843

communication steps, together with their associated forced log writes, are thus eliminated. Despite its efficiency, 1PC has been largely ignored in the implementation of distributed transactional systems. We believe that the main reason for this is due to the strong assumptions made by 1PC protocol designers about transactional processing. For instance, both Coordinator Log (CL) and Implicit Yes-Vote (IYV) protocols assume that participants in a transaction externalize their log records and follow a strict two-phase locking [3] concurrency control. These assumptions are indeed unrealistic in most existing transactional systems. In fact, one might actually ask the question: does 1PC make sense? The aim of this paper is precisely to answer this question.

Through a closer look at 1PC protocols, we draw an interesting parallel between 1PC and 2PC. Whereas 2PC verifies whether or not every participant is able to commit the transaction, the hypothesis underlying 1PC is that every transaction participant that enters the protocol is able to commit the transaction.

With this observation in mind, we revisit 1PC protocols that were proposed in the literature, and dissect the strong assumptions made by their designers about the transactional systems. We present techniques that circumvent those constraints, and make 1PC protocols indeed realistic and useful in practice. In particular, we present a logical logging recovery procedure to eliminate the need for externalizing log records, and we show how to adapt 1PC to weak levels of isolation.

The rest of the paper is organized as follows. Section 2 describes a general model of a distributed transactional system. Section 3 recalls the basic 2PC and its optimizations. Section 4 identifies the assumptions usually made by 1PC designers. Section 5 discusses the assumption related to recovery management. Section 6 discusses the assumption related to the concurrency control mechanism. Finally, Section 7 summarizes the main contributions of this paper.

2. System model

2.1. Sites and processes

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 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 *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.

A process is said to be *correct* if eventually it does never crash. Otherwise, the process is said to be *faulty*. It is important to notice that the period of interest of these definitions is the duration of the commit protocol (i.e., a process is correct if eventually it does never crash until the commit protocol is solved).

We also assume that if a message is sent by a process P_i to a process P_k , then the message is eventually received by P_k , if P_i and P_k do not crash.

To simplify commit problem definitions and algorithm descriptions in the rest of the paper, we assume a perfect failure detection mechanism in the sense that every faulty process is eventually suspected by all processes, and no correct process is ever suspected [4]. Assuming some degree of failure detection unreliability would be possible along the lines of [9], but would add some complexity in the problem definitions and algorithm descriptions. We believe that the issue of unreliable failure detection is out of the scope of this paper.

2.2. Distributed transactions

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 participants. The portion of a transaction executed at one participant is called a transaction branch. To ensure transaction Atomicity, all participants must agree on a unique outcome (Commit or Abort) for all the transaction branches, through an Atomic Commit Protocol (ACP). Although we consider in the following centralized ACPs where a designated participant (e.g, the one that started the transaction) acts as the coordinator in order to orchestrate the actions, the techniques we propose in this paper can be perfectly adapted to a more decentralized context in which all participants assume the same role.

3. Two-phase commit: background

In this section, we briefly recall the basic 2PC protocol together with its optimizations.

3.1. 2PC

As its name indicates, 2PC [7] is formed of two phases: a Voting phase and a Decision phase (Figure 1). During 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. A Yes (or Ready) vote indicates that the local execution of the transaction branch was successful and the participant is able to make its updates permanent, even in the presence of failures. In other words, the participant can locally guarantee the ACID properties of its transaction branch. This participant is said to be in a Prepared state. 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 cannot guarantee one or more of the ACID properties of its transaction branch.

A participant that votes No can unilaterally abort its transaction branch. If a participant votes Yes, it can neither commit nor abort unless it receives the final decision from the coordinator. 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 sends back an acknowledgment. This acknowledgment is a promise from the participant that it will never ask the coordinator about the outcome of the transaction. Once all the acknowledgments are received, the coordinator can forget about the transaction.

The resilience of 2PC to failures is achieved by recording the progress of the protocol in the logs of the coordinator and the participants. Each participant uses its private log to recover from a crash. 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). Indeed, the coordinator force-writes its decision before sending it to the different participants. This ensures that the decision is not lost in case of a coordinator crash. Each participant force-writes (1) its Vote before sending it to the coordinator, and (2) the final decision before acknowledging the coordinator. Actually, a participant that votes Yes force-writes its Vote together with all the updates performed on behalf of the transaction. This

ensures that the participant is able to make its updates permanent even if it crashes (i.e., to ensure transaction Durability). The number of force-writes performed during the execution of the protocol is very important since it determines the number of blocking I/O required for a good behavior of the protocol.

Figure 1 : The basic 2PC protocol.

Figure 1 describes the protocol execution between three participants P1, P2 and P3, where P1 acts as the coordinator of the protocol. Assuming that n is the total number of participants (including the coordinator), 2PC requires 3 communication steps (Request-for-vote, Vote and Decision) and $2n+1$ forced log writes in order to commit/abort a transaction in the absence of failures. As mentioned earlier, this introduces a considerable latency in the system, which increases the transaction response time. The higher the latency of an ACP, the longer the length of time a transaction may be holding shared objects, thus preventing other transactions from accessing these objects. The underlying cost of 2PC has motivated many researchers to propose different optimizations to the basic 2PC.

3.2. 2PC optimizations

The basic 2PC requires information to be explicitly exchanged and logged whether the transaction is to be committed or aborted. This is why it is often called the *Presumed Nothing* protocol (PrN). Several 2PC optimizations that make some presumptions about missing information were proposed.

The *Presumed Abort* optimization (PrA) [10] 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. In a similar way, 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) [10] 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.

One way to reduce communication further is to combine the *Voting phase* with the execution of the transaction. The *Early Prepare* protocol (EP) [12] combines this idea with a PrC: each participant is required to enter the *Prepared* state after executing and before acknowledging every operation. This eliminates the need for the *Request-for-vote* message since the coordinator can interpret the acknowledgment of each operation as a *Ready* vote. However, this optimization introduces a new overhead in the system: each participant is required to force-write its log after the execution of each operation, thus generating one blocking I/O per operation.

4. Understanding one-phase commit

This section recalls 1PC basic idea and compares the performances of 1PC variants with the performances of the 2PC variants recalled in section 3, namely 2PC, PrA, PrC and EP. We then give a closer look to 1PC and identify the major assumptions underlying it.

4.1 A first glance overview

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. Figure 2 compares the different protocols in terms of latency and message complexity needed to commit a transaction². 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. EP requires even fewer messages than PrC and eliminates two communication steps but introduces an important logging overhead during transaction execution. Since the latency of an ACP is a deciding factor in the performance of distributed transactions, different authors have proposed the use of One-Phase Commit protocols (1PC) that reduce both the number of forced log writes and communication steps without additional overhead. In particular, we know of at least two variations of 1PC: the Coordinator Log (CL) protocol

of Stamos and Cristian [12], and the Implicit Yes-Vote (IYV) protocol of Al-Houmaily and Chrysantis [1].

Roughly speaking, 1PC reduces the latency of 2PC by eliminating its Voting phase which is the mean by which the coordinator of 2PC verifies whether or not participants in the transaction can locally guarantee the ACID properties of their transaction branches. The governing idea of 1PC is to eliminate the need for this verification during the protocol execution by having these properties already guaranteed at commit time at every participant. Obviously, this introduces some assumptions on the way participants manage transactions. The main difference between the various versions of 1PC has to do with the way recovery is managed.

	Nb. of Messages	Latency Forced Log Writes	Latency Communication Steps
2PC	4 (n-1)	1 + 2n	3
PrA	4 (n-1)	1 + 2n	3
PrC	3 (n-1)	2 + n	3
EP	(n-1)	1 + (n+n_op)	1
CL	(n-1)	1	1
IYV	2 (n-1)	1 + n	1

n_op denotes the number of operations performed by a transaction.

Figure 2. The cost of transaction commit in various ACPs

As conveyed by Figure 2, 1PC variants are very efficient compared to 2PC protocols, in terms of message complexity, number of forced log writes and number of communication steps.

4.2 A closer look to 1PC

Variations of 2PC guarantee transaction Atomicity 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 operations are terminated. 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.

The basic assumption underlying 1PC is that a participant does not need to vote. This does not mean however that the transaction's outcome is known in advance. *Commit* is decided if all operations have been acknowledged and no failure occurs, and *Abort* might be decided otherwise. However, in most cases (i.e., commit cases), the coordinator has just to send a single message to the participants asking them to commit.

² This corresponds to the most frequent case since most transactions are expected to commit in the absence of failures.

Unlike 2PC, participants here do not vote. In other words, 1PC does not take care of also ensuring the ACID properties of the corresponding local branches of the transaction. This means that before triggering the commit protocol, the coordinator makes sure that these properties are locally ensured at all participants. In other words, the coordinator acts as a nice dictator and makes sure that no participant can have any reasonable reason to vote *No*. This observation leads to better understand the assumptions usually made (explicitly or implicitly) by 1PC protocols. More precisely:

1. 1PC protocols [1, 12] assume that all the transaction operations have been acknowledged (i.e., all operations have been successfully executed till completion) before the protocol is launched. This means that the Atomicity of all the local transaction branches (i.e., local Atomicity) is already ensured at commit time.

2. 1PC protocols assume that integrity constraints are checked after each update operation and before acknowledging the operation. Thus, Consistency is ensured for all the local transaction branches 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 protocols assume that all participants serialize their transactions using a pessimistic concurrency control protocol that avoids cascading aborts (i.e., strict two-phase locking [3]). In this context, a transaction that executes successfully all its operations can no more be aborted due to a serialization problem. This actually means that serializability (Isolation) of all the local transaction branches is already ensured at commit time (e.g., optimistic concurrency control protocols that check serializability at commit time are excluded).

4. 1PC protocols assume that, at commit time, the effects of all the local transaction branches are logged on stable storage, and hence the Durability property is ensured. In the Coordinator Log protocol, 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. 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 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) [6] whenever they acknowledge an update operation, and the list of read locks granted during the execution of a read 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.

Note that in CL and IYV, a coordinator is required to force-write its log records prior to sending a decision message to the different participants. On receipt of the final decision, participants in the IYV protocol are required to force-write a *Commit* log record since they are required to acknowledge this decision.

4.3 1PC drawbacks

By eliminating the need for votes, 1PC protocols indeed achieve better performances than 2PC protocols. However, 1PC protocols introduce strong assumptions on the way participants manage transactions. 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 [5] and OTS from OMG [11] assume the same behavior. On the contrary, (a) the recovery protocols assumed increase the size of acknowledgment messages and violate *site autonomy*³ by forcing participants to externalize their local log records, and (b) the assumption of strict two-phase locking might also appear very strong as several database systems ensure lower levels of isolation standardized by SQL2 [13] and do not guarantee serializability.

In the rest of the paper, we dissect each of these two drawbacks (i.e., (a) and (b)) respectively in sections 5 and 6, and we show how they can be avoided or weakened.

³ Site autonomy means, among others, that internal information concerning the local execution of transactions (e.g., LSN, log records, etc.) remains private to a site and cannot be exported and exploited by the distributed transactional system. Preserving site autonomy is crucial in practice as commit protocols must often be used with existing commercial transactional systems and should not require changes to these systems.

Figure 3: Scenario of 1PC with logical logging.

5. Recovery

In this section, we propose a *logical logging* mechanism that ensures correct recovery. However, unlike 1PC protocols that have been proposed so far, our mechanism preserves site autonomy and does not increase communication cost. Section 5.1 introduces our logical logging idea. Section 5.2 sketches the behavior of a 1PC protocol relying on logical logging. Section 5.3 introduces the associated recovery algorithm, and finally, Section 5.4 proves its correctness.

5.1 The assumption of log externalization

Coordinator Log and Implicit Yes-Vote violate site autonomy by forcing participants to externalize their redo logs. To circumvent 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 One-Phase Commit, the failed transaction branches will be re-executed using the operations registered in the coordinator's log.

Logical logging at the coordinator 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).

However, logical logging makes the recovery protocol more tricky. The problem is to show that, in a redo procedure, 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.

5.2 Logical logging: basic scenario

The typical scenario of a 1PC protocol relying on our logical logging mechanism is depicted in Figure 3. It is important to note that our mechanism is generic in the sense that it can be applied to all 1PC protocols. To simplify the figure, we show the actions executed by a single participant P_k .

A local *Agent* (called $Agent_k$) is associated to each participant P_k for recovery purposes. 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. In the following, T_{ik} denotes the local branch of transaction T_i executed at participant P_k .

In Figure 3, Step 1 represents the sequence of operations executed on behalf of T_{ik} . The coordinator registers in its log each operation that is to be executed.

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 P_k . Every operation is acknowledged up to the application.

Step 2 corresponds to the termination of the transaction from the application point of view. Once all the acknowledgments are received by the application, it issues a commit request. At this point, properties ACI are locally guaranteed by the participants for all the local transaction branches. However, since participants are not aware of the termination of the transaction, they cannot guarantee the Durability property. Durability is ensured by the coordinator itself, which force-writes all T_i 's log records on stable storage.

Once Step 2 is achieved, the ACID properties are guaranteed altogether for all the local transaction branches. Thus, the coordinator takes a *Commit* decision and force-writes this decision on stable storage (Step 3). Note that, in order to improve performances, T_i 's log records (Step 2) together with the *Commit* decision (Step 3) can be forced on stable storage at the same time, thereby generating a single blocking I/O. Then, the coordinator broadcasts the *Commit* decision to all participants and waits for their acknowledgments. When all acknowledgments are received, the coordinator forgets the transaction. The role of the *Agent* during this step will be detailed in the next section, as it has to do only with recovery.

If transaction T_i is to be aborted, Step 2 and Step 3 are simpler than their committing counterparts. The coordinator discards all T_i 's log records and broadcasts an *Abort* decision message to all participants. We assume a presumed abort protocol, so abort messages are not acknowledged and the *Abort* decision is not recorded in the coordinator's log.

5.3. Logical logging: recovery

Figure 4 details the recovery algorithm executed in case a participant crashes. For the sake of clarity, step numbers correspond to the steps ordering.

Step 1 and Step 2 represent the standard recovery procedure executed by a crashed participant P_k . To preserve site autonomy, we make no assumptions on the way these steps are handled. However, as mentioned in Section 2.2, each participant is assumed to guarantee the local ACID properties of all transaction branches submitted to it. Step 3 is necessary to determine if the k^{th} branch of some globally committed transactions have to be locally re-executed by the crashed participant.

Participant's Algorithm

- 1- forward recover the transactions that reached their commit state before the crash.
- 2- backward recover all other transactions
- 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 $(\text{commit}_i) \notin \text{coordinator's log}$, then
broadcast (abort_i) to all other T_i participants
and forget T_i
- 5- if $(\text{commit}_i) \in \text{coordinator's log}$, then
if $\text{ack}_k(\text{commit}_i) \notin \text{coordinator's log}$, 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 4: Recovering a participant crash.

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 backward recovered during Step 2, or T_{ik} is locally 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 of this step, we assume for the moment that the coordinator can query a participant to learn the exact status of T_{ik} (Step 6). We detail afterwards the way we handle this, without violating site autonomy.

The participant answers during Step 7. If T_{ik} has been successfully committed by P_k , the coordinator does nothing. Otherwise, T_{ik} has been backward recovered during Step 2 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 status of its local transaction branches. As introduced earlier, our solution relies on a local *Agent* associated to each 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. During Step 3 of the scenario presented in Figure 3, the coordinator broadcasts the *Commit* decision to each participant via its *Agent*. When it receives the decision, the *Agent* issues an additional operation "write record $\langle \text{commit}_i \rangle$ " on behalf of the local transaction branch it is in charge of (e.g., T_{ik}).⁴ 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. The rest of the protocol remains unchanged. 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 the existence of record $\langle \text{commit}_i \rangle$ 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 $\text{write} \langle \text{commit}_i \rangle$ is performed on behalf of T_{ik} . Otherwise, T_{ik} has been backward recovered during Step 2 and must be re-executed.

5.4. Logical logging: correctness

The recovery algorithm associated to an ACP protocol must guarantee the Atomicity property of distributed transactions even in the presence of failure. This means that once the *Commit* decision has been taken by the coordinator, all the updates performed by a failed transaction branch can be recovered. In the traditional 2PC protocol, each participant votes *Yes* during the first phase of the protocol only if it can enter a *Prepared* state, that is it has logged enough information to be able to recover the local transaction branch. 1PC protocols eliminate the first phase of the ACP, precluding

participants from entering a *Prepared* state. Consequently, the Atomicity of transactions relies now on the information logged in the coordinator's log. In *Coordinator Log* and *IYV* protocols, 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. However, if the coordinator exploits logical logging rather than physical logging, the recovery protocol 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 crash 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. By definition, each participant ensures locally the ACID properties of each transaction branch submitted to it. Assume the coordinator has taken the decision to commit T_i and then P_k crashes. The Atomicity property ensures that if P_k crashed before the commit of T_{ik} was achieved, T_{ik} is backward recovered during Step 2 of the recovery algorithm and the record $\langle \text{commit}_i \rangle$ is discarded⁵. Otherwise (i.e., P_k crashed after the commit of T_{ik}), the Durability property guarantees the presence of the record $\langle \text{commit}_i \rangle$ at P_k . These two situations are differentiated during Step 6 of the recovery algorithm. Step 8 of the algorithm forward recovers only transaction branches that have been locally aborted. Thus, 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 in the same order as they have been logged on the coordinator, that is 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. Indeed, most transactional systems execute transactions in parallel through several threads of control and no assumption can

⁴ Note that this operation never generates a dependency cycle (i.e., deadlock) since it is the last operation executed in any transaction to be committed.

⁵ We recall that the operation $\text{write record } \langle \text{commit}_i \rangle$ is performed on behalf of T_{ik} .

be made on their local scheduling policy, without violating *site autonomy*. 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, whatever is the order of execution of these operations, the local 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.

$\varphi = \{T_{ik} / \text{commit}_i \in \text{coordinator's log} \wedge \text{ack}_k(\text{commit}_i) \notin \text{coordinator's log} \wedge \langle \text{commit}_i \rangle \notin P_k\text{'s state}\}.$

Since P_k uses a rigorous serialization protocol, $\forall T_{ik}, T_{jk} \in \varphi, \neg \exists (T_{jk} \rightarrow T_{ik})$, where $\rightarrow$ represents a precedence in the serialization order. Otherwise, T_{ik} would have been blocked during its first 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} do not compete on the same resources. In addition, 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. The local re-execution of all other transaction branches belonging to φ cannot change this state since they do not compete on the same resources. 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, T_{ik} is guaranteed to re-access the same state of its resources and consequently, generates the same local state as after its initial execution.

6. Concurrency control

In this section, we show that the strict two-phase locking assumption can be relaxed, making 1PC compatible with database systems supporting SQL2 levels of isolation.

6.1 Strong two-phase locking

Coordinator Log and IYV make strong assumptions on the serialization policy used by the participants. The goal of this section is to determine the class of participants that match these assumptions in order to answer the question: are they exotic or of general interest? To ensure a correct behavior of the Coordinator Log and IYV protocols,

participants must satisfy the two following properties:

P1: If all operations of T_{ik} have been acknowledged by P_k , then T_{ik} is serializable on P_k .

P2: Local schedules are cascadeless

It follows that a transaction that executes successfully all its operations can no more be aborted due to a serialization problem. Optimistic concurrency control protocols that check serializability at commit time [3] satisfy P2 but not P1. Conversely, timestamp ordering protocols [3] satisfy P1 but not P2. Only rigorous concurrency control protocols [2] satisfy both P1 and P2.

A serialization protocol is said to be rigorous if, for all pairs of transactions T_i and T_j , if a conflict occurs and generates a dependency of the form $T_i \rightarrow T_j$, then T_j is blocked until T_i terminates. A typical example of a rigorous concurrency control protocol is the strict two-phase locking (2PL) protocol. Fortunately, 2PL is the most widely used protocol for ensuring serializability of transactions.

6.2 Levels of isolation

In this section, we consider database systems case, which represent the dominant part of commercial transactional systems. Commercial database systems are likely to use levels of isolation standardized by SQL2 [13] in combination with 2PL. We recall below the SQL2 levels of isolation and we 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. That is, successive reads of the same object give always the same result but successive 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.

Read Uncommitted: Transactions running at this level may do dirty reads. For this reason, they are not allowed to update the database.

Isolation levels are widely exploited because they allow faster executions, increase transaction parallelism and reduce the risk of deadlocks. Although non-serializable schedules may be produced, they are considered as semantically correct. Database systems

supporting isolation levels satisfy, instead of property P1 above, the following property P3:

P3: If all operations of T_{ik} have been acknowledged by P_k , then T_{ik} cannot be rejected by P_k due to a serialization problem.

Indeed, either no operation of T_{ik} generates a conflict, or these conflicts are ignored by the local serialization policy. Property P3 still complies with a 1PC protocol. The problem is to ensure that P3 is strong enough to enforce the correctness of the recovery algorithm. This proof is straightforward if the coordinator's log contains physical redo records, like in *Coordinator log* or *IYV*. The redo log records are reinstalled 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. In the following, we prove that the recovery algorithm associated to logical logging (cf. section 5) remains valid. We begin by studying the schedules that may be produced depending on T_i 's isolation level.

Serializable: If all transactions run at this level of isolation, the schedules produced remain rigorous. Thus, the recovery algorithm proposed in section 5 remains valid.

Repeatable Read: This level of isolation ignores some Read/Write conflicts. Let Select_{ik} denote a selection performed by T_{ik} that retrieves all objects satisfying predicate Q and Insert_{jk} the creation by T_{jk} of a new object satisfying Q . Although a phantom is produced, the following local schedule is accepted without blocking: $\text{Select}_{ik} < \text{Insert}_{jk} < \text{Commit}_{jk} < \text{Select}_{ik} < \text{Commit}_{ik}$, where "<" represents the precedence of events. Assume P_k crashes and T_{ik} needs to be forward recovered. Since T_{ik} is re-executed after the commit of T_{jk} , the schedule produced is equivalent to: $\text{Insert}_{jk} < \text{Commit}_{jk} < \text{Select}_{ik} < \text{Select}_{ik} < \text{Commit}_{ik}$ and differs from the initial one.

Read Committed: This level of isolation ignores Read/Write conflicts that occur on a same object O . Thus the following local schedule is accepted without blocking: $\text{Read}(O)_{ik} < \text{Write}(O)_{jk} < \text{Commit}_{jk} < \text{Commit}_{ik}$. If P_k crashes and T_{ik} needs to be forward recovered, the schedule produced is equivalent to: $\text{Write}(O)_{jk} < \text{Commit}_{jk} < \text{Read}(O)_{ik} < \text{Commit}_{ik}$ and differs from the initial one. Note that the following initial schedule is also accepted without blocking: $\text{Read}(O)_{ik} < \text{Write}(O)_{jk} < \text{Commit}_{jk} // \text{Commit}_{ik}$ (where $//$ represents a parallel execution of two operations). If P_k crashes, it may happen that both T_{ik} and T_{jk} need to be forward

recovered. The recovery could produce the same wrong schedule as before and can even generate a blocking if T_{ik} attempts to read object O just after T_{jk} modifies it. If blocking is possible during recovery, deadlocks become possible too, if we consider a larger sequence of operations.

Read Uncommitted: Transactions running at this level are not allowed to update the database. They are qualified as *Read Only* transactions that do not have to be re-executed in case of crash. *Read Only* transactions represent the most propitious case for any ACP.

Thus, *Repeatable Read (RR)* and *Read Committed (RC)* isolation levels may introduce problems in the recovery protocol. In the examples given above, no inconsistencies are introduced in the database. However, inconsistencies may appear if the result of the read or select operations are used in subsequent updates (e.g., T_{ik} reads O and then executes $O'=O$). A straightforward solution could be to preclude updates for all transactions running with an isolation level lower than *Serializable*. A more permissive approach is possible with little effort. For each transaction T_i running in *RR* or *RC* isolation levels, the correctness of the recovery algorithm can be re-established if the coordinator enforces the following rule:

$(\text{Isolation}(T_i) = \text{RR or RC}) \text{ and } (T_i \text{ overlaps } T_j) \Rightarrow \forall k$
 $[\text{receive}(\text{ack}(\text{commit}_{ik}))] < [\text{send}(\text{commit}_{jk})]$

This means that the coordinator serializes the commit requests for all couple of transactions, if one of them runs in a low isolation level and if their executions overlap on a given participant. Except during the commit, T_i and T_j are allowed to execute in parallel, thereby preserving the benefit of low levels of isolation. Actually, this forced commit ordering corresponds to the conflicts of the form $(T_i \xrightarrow{\text{RW}} T_j)$ that may be ignored by participant P_k . Thus, T_i is guaranteed to commit before T_j so that they are never forward recovered in parallel.

As a conclusion, 1PC with physical logging is compatible with any participant that serializes transactions with a rigorous protocol, even if levels of isolation are exploited. This embraces almost all today's commercial database systems. 1PC with logical logging can be adapted to this same class of database systems with little effort.

7. Conclusion

This paper identifies the circumstances under which One-Phase Commit (1PC) makes sense in distributed transaction processing. We first draw the observation

that, in contrast to Two-Phase Commit (2PC) which enables every participant in a transaction to abort if the ACID properties of the local transaction branches are not guaranteed, 1PC makes the assumption that these properties are guaranteed at commit time for every local transaction branch. In other words, the ACID properties of local transaction branches are supposed to be ensured before the 1PC protocol is triggered. With this observation in mind, we explain the assumptions underlying 1PC protocols that were proposed in the literature, and we discuss the way these assumptions can be weakened. In particular, we show that the assumption of log externalization usually made by 1PC protocols designers is not really needed and we present a logical logging protocol that does not require access to internal logs (i.e., does not violate site autonomy). We also show that one can adapt 1PC to a concurrency control that simply ensures some level of isolation [13], and does not follow the strict two-phase locking rule.

References

- [1] Y. Al-Houmaily and P. Chrysanthis. Two-phase Commit in Gigabit-Networked Distributed Databases. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems*, September 1995.
- [2] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, 1(2), 1992.
- [3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.
- [4] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), 1996.
- [5] CAE Specification, *Distributed Transaction Processing: the XA Specification*, XO/CAE/91/300, 1991.
- [6] J. Gray and A. Reuter. *Transaction processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [7] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*. R. Bayer, R.M. Graham and G. Seegmuller editors, LNCS 60, Springer Verlag, 1978.
- [8] J. Gray. A Comparison of the Byzantine Agreement Problem and the Transaction Commit Problem. *Fault-Tolerant Distributed Computing*. B. Simons and A. Spector editors, LNCS 448, Springer Verlag, 1987.
- [9] R. Guerraoui. Revisiting the Relationship between Non-Blocking Atomic Commitment and Consensus. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, LNCS 972, Springer Verlag, September 1995.
- [10] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986.
- [11] *Object Transaction Service*, OMG Document 94.8.4, OMG editor, August 1994.
- [12] J. Stamos and F. Cristian. A Low-Cost Atomic Commit Protocol. In *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems*, October 1990.
- [13] International Standardization Organization IS, *Information Processing Systems - Database Language SQL*, ISO/IEC 9075, 1992.