Atomic commit protocols, their integration, and their optimisations in distributed database systems

Yousef J. Al-Houmaily

Department of Computer and Information Programs,
Institute of Public Administration,
Riyadh 11141, Saudi Arabia
E-mail: houmaily@ipa.edu.sa
E-mail: ysf_al@yahoo.com

Abstract: Advanced software application systems commonly execute atop multiple interconnected computing platforms. Regardless of whether the underlying platforms being homogeneous or heterogeneous, such systems require basic reliability guarantees to ensure deterministic outcomes in the presence of faults. Similar guarantees are provided by database systems through the 'atomicity' property of transactions. In distributed database systems, this property is ensured, across different database sites, by means of 'atomic commit protocols' (ACPs). An ACP guarantees, in spite of possible failures, that each transaction has a deterministic final outcome. This outcome represents either the execution of the transaction as a whole, across all participating sites, or none at all. Thus, it is imperative to trace the problem of atomic commitment in distributed database systems and to highlight its current dimensions as well as its proposed solutions. Such an effort establishes a foundational stage for investigating and developing more elaborate and novel solutions to the problem. These solutions are likely to satisfy the reliability needs of future generations' application systems in a cost-effective manner.

Keywords: atomic commit protocols; ACP; atomicity; database recovery; distributed transaction management; integrated database systems; two-phase commit; 2PC.

Reference to this paper should be made as follows: Al-Houmaily, Y.J. (2010) 'Atomic commit protocols, their integration, and their optimisations in distributed database systems', *Int. J. Intelligent Information and Database Systems*, Vol. 4, No. 4, pp.373–412.

Biographical notes: Yousef J. Al-Houmaily received his BSc in Computer Engineering from King Saud University, Saudi Arabia in 1986, MSc in Computer Science from George Washington University, Washington DC in 1990, and PhD in Computer Engineering from the University of Pittsburgh in 1997. He is currently an Associate Professor in Department of Computer and Information Programs at the Institute of Public Administration, Riyadh, Saudi Arabia, where he was the Director from November 1997 till June 2002. He provided consultation services to a number of government agencies in Saudi Arabia including the Deputy Prime Minister's Office, the Control and Investigation Board and the General Presidency for Girls Education. From July 2005 till June 2006, he was a Visiting Assistant Professor at the School of Computer Science, University of Waterloo, Canada. His current research interests are in the areas of database management systems, mobile distributed computing systems and sensor networks.

1 Introduction

Continuous advancements in intranet and internet technologies enable for the development of a more and more advanced software application systems. Multi-organisational workflows and web services are currently two common examples from a long list of such application systems. These systems are commonly built atop multiple interconnected computing platforms. Regardless of whether the underlying platforms being homogeneous or heterogeneous, such systems require basic reliability guarantees to ensure deterministic outcomes in the presence of faults. This requirement, given the continuous technological advancements, is expected to be in a more progressive demand by future generations' application systems.

In database systems, similar reliability guarantees are provided by the 'atomicity' property of transactions. This property means that each transaction is executed as a single indivisible (i.e., atomic) unit of work even in the presence of faults. That is, a transaction either *commits* (i.e., executes to completion successfully), reflecting all its effects on the state of the database; or *aborts* (i.e., fails at some point during its execution), nullifying any of its propagated effects from the state of the database.

Assuming that each database site preserves atomicity of transactions at its local level, in distributed database systems, *global atomicity* cannot be guaranteed without taking additional measures. This is because a distributed transaction might end up committing at some participating sites and aborting at others due to a site or a communication failure. This jeopardises global atomicity and, consequently, the consistency of the (distributed) database. For this reason, there is a need to ensure atomicity at the global level (across all participating sites) besides ensuring it at the level of each individual database site. This is the essence of any *atomic commit protocol* (ACP).

An ACP is basically a synchronisation protocol that ensures global atomicity of transactions in spite of possible site and communication failures. Specifically, it guarantees a deterministic final outcome for each distributed transaction whereby a transaction is either performed, across all participating database sites, to its completion or not at all. In fact, ACPs provide a solution to the *problem of atomic commitment* in distributed database systems, a special case of the problem of *consensus in the presence of faults*. The latter problem is known in distributed systems as the *Byzantine Generals* problem (Lamport et al., 1982) and is not solvable, in its most general case, without some simplifying assumptions. In distributed database systems, ACPs solve the problem under the following general assumptions (among others that are sometimes ACP specific):

1 *Each site is sane*: a site is fail-stop where it never deviates from its prescribed protocol. That is, a site is either operational or not but never behaves abnormally causing *commission* (i.e., intentionally misleading) failures.

2 *Eventual recovery*: a failure (whether site or communication) will be, eventually, repaired.

3 *Binary outcome*: all sites unanimously agree on a single binary outcome, either commit or abort.

Although ACPs are the only means to ensure atomicity in distributed database systems, ACPs are known for their adverse effects on the overall systems' performances for two reasons. Firstly, an ACP consumes, during normal system operation, a substantial amount of a transaction's execution time due to its required synchronisation activities

(Spiro et al., 1993). These activities include exchanging a number of messages among involved sites and accessing the, relatively slow, stable storage of each individual site. Secondly, ACPs are known for their *blocking*, in the case of a failure, which means that a participant in a transaction's execution might not be able to unilaterally determine the final status of the transaction in the presence of the failure. Thus, rendering all resources held by the pending transaction at the participant's site unusable by other concurrently executing transactions until the final status of the transaction is resolved. To minimise the above two effects, a variety of ACPs and optimisations have been proposed in the literature and the research in this area continues to be an important topic for many distributed database environments. These include main memory databases (e.g., Lee and Yeom, 2002), mobile database systems (e.g., Nouali-Taboudjemat et al., 2007), real-time databases (e.g., Haritsa et al., 2000), active-networks (e.g., Awan and Younas, 2004) and component-based architectures (e.g., Rocha et al., 2007); besides traditional (homogenous) distributed databases (e.g., Al-Houmaily, 2005).

A second important issue related to ACPs, besides performance, is interoperability among database sites that adopt different ACPs. This issue arises in environments such as multi-database systems and the internet where different database sites might use different ACPs yet, wish to collaborate with each other to accomplish specific business tasks. In this case, *incompatibilities* among the used ACPs arise and might become a barrier in the face of such collaborations if not resolved in a practical manner.

Advanced software application systems require reliability guarantees similar to the one's available for distributed database systems. For this reason, Organization for the Advancement of Structured Information Standards (OASIS) *Web Services Transaction* (WS-T) (OASIS, 2007), the currently most widely accepted web service industry specification, provides atomicity-preserving supports for such systems. Similar supports are also commonly integrated into middleware platforms including Sun Microsystems' Java Enterprise Edition (JEE) (Sun Microsystems, 2006), Object Management Group CORBA Transaction Service Specification (Object Management Group, Inc., 2003) and Microsoft's .Net Framework (Microsoft Corporation, 2007). Thus, it is imperative to trace the problem of atomic commitment in distributed database systems and to highlight its current dimensions as well as its proposed solutions. Such an effort establishes a foundational stage for investigating and developing more elaborate and novel solutions to the problem. These solutions are likely to satisfy the reliability needs of future generations' application systems in a cost-effective manner. Such an effort is also anticipated to add further stimulation to this ever-evolving area of research.

The rest of this paper is structured as follows: Section 2 contains a comprehensive treatment to the basic *two-phase commit* (2PC) protocol (Gray, 1978; Lampson, 1981) being the first known and published ACP, and being the base for the design of all the other ACPs. It also discusses the main issues and metrics that are commonly used for the evaluation of ACPs. This latter discussion leads to the presentation of two classes of ACPs: *efficiency-geared* and *reliability-geared*. The first class addresses ACPs that were proposed to enhance the performance of commit processing during normal system's operation. This class of protocols represents the topic of Section 3. The second class addresses ACPs that were proposed to enhance reliability by reducing the effects of blocking. This class of protocols represents the topic of Section 4. The incompatibilities among ACPs are discussed in Section 5. The discussion highlights the sources of incompatibilities among ACPs and the most recent proposals to resolving them. Section 6 starts by distinguishing between ACP variants and ACP optimisations. Then, it presents the most widely known ACP optimisations. Section 7 summarises the paper and provides some concluding remarks.

2 Distributed transactions and two-phase commit

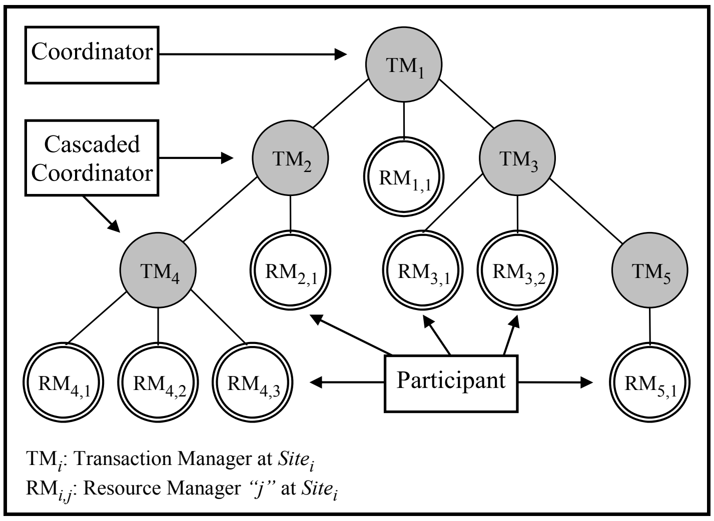
A distributed transaction accesses data stored at different database sites that are interconnected via a communication network. Each database site is assumed to consist of a *transaction manager* (TM) and one or more *resource managers* (RMs). A TM at a site is responsible for different management and coordination activities. These include assigning identifiers to transactions, monitoring their progress, supervising their completion and coordinating failure recovery. On the other hand, RMs are responsible about performing the actual work on data. A RM could be any system that manages shared data and maintains (local) data consistency even in the case of failures. In this article, a RM is assumed to be a relational database management system although it could be any other transactional system such as a transactional queue or a transactional file system.

When a distributed transaction starts executing at a site, the TM at the *originating* site of the transaction assigns to the transaction a system-wide unique identification number. The TM, at the originating site of the transaction, also informs all local RMs about the start of the transaction. In this way, the local TM and all local RMs are aware of the existence of the transaction. Alternatively, a RM could have the ability to *dynamically* register itself, as a participant in the execution of the transaction, with its local TM once it receives some database operation(s) for execution from the transaction. In this latter case, there is no need for the TM to inform such a RM about the start of the transaction once it begins executing at the site.

Dynamic registration of a RM with its (local) coordinating TM could be accomplished through a 'Join' method similar to the one available in the X/Open distributed transaction processing standard (X/Open Company Limited, 1996) or any alternative method. The details of the mechanism by which RMs join in the execution of a transaction are not particularly important so long as each TM is aware of all local RMs that participate in a transaction's execution. A TM needs to know all local RMs that participate in the execution of each transaction so that it can perform its coordination responsibility which guarantees unique final outcome across all of them.

The above execution model of distributed transactions is called *flat* or *two-level transaction execution* model whereby the coordinating TM, conceptually, resides at the top (i.e., first level) of the execution tree while the participating RMs reside at the leaves (i.e., the second level). A more general model called *multi-level transaction execution* (MLTE) or *tree of processes* model is commonly used in practice. In this latter model, a distributed transaction could *fork* (or *spawn*) an execution branch that consists of operations that execute at a remote database site and the remote site could fork another execution branch and so on. Similar to the flat execution model, a TM at a remote site that executes some of a transaction's operations has to be aware of all local RMs that join in the execution of the transaction. Thus, a tree hierarchy, similar to the one depicted in Figure 1, is formed for each distributed transaction whereby the TM at each participating site is aware of all local RMs that participate in the execution of the transaction as well as all neighbouring (remote) TMs.

Figure 1 A typical execution tree

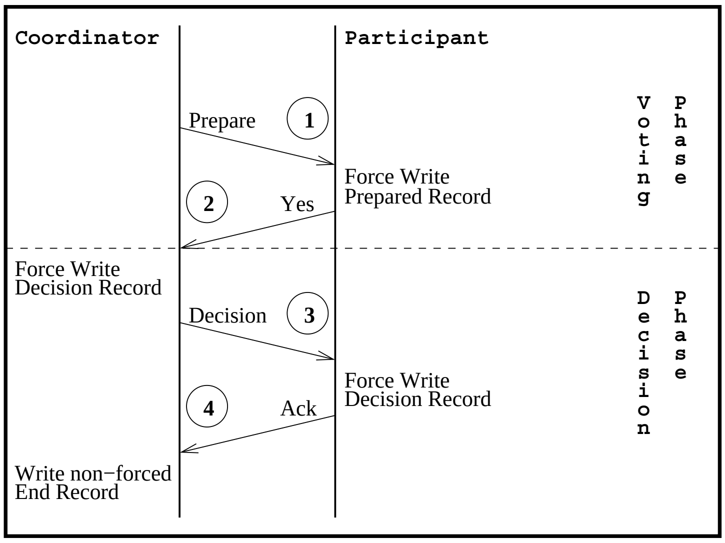


In either of the two execution models above, once a transaction finishes its execution and indicates its termination point, through a commit primitive, the orchestration for the final outcome of the transaction begins. The final outcome is reached by the participants by playing different roles and performing different actions in the execution of the transaction. Specifically, in 2PC, each transaction is associated with a designated TM called the *coordinator*, *superior* or *master*. Although the coordinator of a transaction could be any of the TMs that participate in the transaction's execution, which is the case in *peer-to-peer* environments, the coordinator is commonly the TM at the originating site of the transaction (i.e., the TM at the site where the transaction is first initiated). The rest of the participants are called *subordinates*, *cohorts*, *slaves* or, simply, *participants*. A participant could be an internal node representing a remote TM or a leaf participant representing a RM. If the participant is a remote TM, it is called *cascaded coordinator* because it acts as a participant with respect to its direct ancestor, in the transaction execution tree, and a coordinator with respect to its direct descendant(s). Figure 1 depicts the relationships among the different participants and their used naming conventions in the rest of this article. Assuming a two-level transaction execution tree, the orchestration of 2PC is performed as explained in the following section.

2.1 The basis two-phase commit protocol

As the name implies, 2PC consists of two phases, namely a *voting phase* and a *decision phase*, as shown Figure 2. During the voting phase, the coordinator requests all the participants in the transaction's execution to *prepare-to-commit* whereas, during the decision phase, the coordinator either decides to commit the transaction if *all* the participants are prepared to commit (voted 'yes'), or to abort if any participant has decided to abort (voted 'no'). On a commit decision, the coordinator sends out commit messages to *all* participants whereas, on an abort decision, it sends out abort messages to *only* those (required) participants that are prepared to commit (voted 'yes').

Figure 2 The two-phase commit protocol


Coordinator
Participant
Prepare
1
Force Write Prepared Record
2
Yes
Voting Phase
Force Write Decision Record
Decision
3
Force Write Decision Record
4
Ack
Decision Phase
Write non−forced End Record

When a participant receives a prepare-to-commit message for a transaction, it validates the transaction with respect to data consistency. If the transaction can be committed (i.e., it passed the validation process), the participant responds with a 'yes' vote. Otherwise, the participant responds with a 'no' vote and aborts the transaction, releasing all the resources held by the transaction.

If a participant had voted 'yes', it can neither commit nor abort the transaction unilaterally and has to wait until it receives a final decision from the coordinator. In this case, the participant is said to be *blocked* for an indefinite period of time called *window of uncertainty* (or *window of vulnerability*) awaiting the coordinator's decision. When a participant receives the final decision, it complies with the decision, sends back an *acknowledgment message* (Ack) to the coordinator and releases all the resources held by the transaction.

When the coordinator receives Acks from all the participants that had voted 'yes', it *forgets* the transaction by discarding all information pertaining to the transaction from its *protocol table* that is kept in main memory.

The resilience of 2PC to system and communication failures is achieved by recording the progress of the protocol in the logs of the coordinator and the participants. The coordinator force-write a *decision* record prior to sending out its final decision to the participants. Since a *forced write* of a log record causes a flush of the log onto a stable storage that survives system failures, the final decision is not lost if the coordinator fails. Similarly, each participant force-write a *prepared* record before sending its 'yes' vote and a *decision* record before acknowledging a final decision. When the coordinator completes the protocol, it writes a non-forced *end* record in the volatile portion of its log that is kept in main memory. This record indicates that all (required) participants have received the final decision and none of them will inquire about the transaction's status in the future. This allows the coordinator to (permanently) forget the transaction, with respect to the 2PC protocol, and to garbage collect the log records of the transaction when necessary.

2.1.1 Recovery in two-phase commit

Site and communication failures are detected by *timeouts*. When a coordinator or any participant times-out awaiting a message, during the course of 2PC, the timeout is perceived as an indication to a failure. Consequently, a recovery manager is invoked to handle the failure. In 2PC, there are *four* places where a communication failure might occur and the recovery manager is launched to handle it, as indicated in Figure 2.

The first place is when a participant is waiting for a prepare-to-commit message from the coordinator. This occurs before the participant has voted. In this case, the participant may unilaterally decide to abort the transaction. The second place is when the coordinator is waiting for the votes of the participants. Since the coordinator has not made a final decision yet and no participant could have decided to commit, the coordinator can decide to abort. The third place is when a participant had voted 'yes' but has not received a commit or an abort final decision. In this case, the participant cannot make any unilateral decision because it is uncertain about the coordinator's final decision. The participant, in this case, is *blocked* until it re-establishes communication with the coordinator. Once communication is re-established, the participant inquires the coordinator about the final decision and resumes the protocol by enforcing and, then, acknowledging the coordinator's decision. The fourth place is when the coordinator is waiting for the Acks of the participants. Since the coordinator cannot forget the transaction before it receives Acks from all the (required) participants, it re-submits the final decision to those participants that have not acknowledged the decision once it re-establishes communication with them.

To recover from site failures, there are two cases to consider: a coordinator's failure and a participant's failure. For a coordinator's failure, the coordinator, upon its restart, scans its stable log and re-builds its protocol table to reflect the progress of 2PC for all transactions that were pending prior to the failure. Specifically, it has to consider only those transactions that have started the protocol and have not finished it prior to the failure (i.e., transactions that have decision log records without corresponding end log records in the stable log). For other transactions, i.e., transactions that were active at the coordinator's site prior to its failure without a decision record, the coordinator considers them as aborted transactions. Once the coordinator re-builds its protocol table, it completes the protocol for each pending transaction by re-submitting its final decision to all (required) participants whose identities are recorded in the decision record and waiting for their Acks. Since some of the participants in a transaction's execution might have already received the decision prior to the failure and enforced it, completing the execution of the whole transaction at their sites, these participants might have already forgotten that the transaction had ever existed. Such participants simply reply with *blind* Acks indicating that, whatever was the decision, they have already received and enforced it prior to the failure.

For a participant's failure, the participant, as part of its recovery procedure, checks its log for the existence of any transaction that is in a prepared to commit state (i.e., has a prepared log record without a corresponding final decision log record). For each prepared to commit transaction, the participant inquires the transaction's coordinator about the final decision. Once the participant receives the final decision from the coordinator, it completes the protocol by enforcing and, then, acknowledging the coordinator's decision. Notice that a coordinator will be always able to respond to such inquires because it cannot forget a transaction before it has received Acks from all (required) participants.

The basic 2PC that we discussed above is also referred to as the *presumed nothing* (PrN) (Lampson and Lomet, 1993). This is because it treats all transactions uniformly, whether they are to be committed or aborted, requiring information to be explicitly exchanged and logged at all times. However, there is a case where a participant might be in a prepared to commit state and the coordinator does not remember the transaction. This case occurs if the coordinator fails after it has sent out prepare-to-commit messages and just before it made the final decision. In this case, not finding a decision record for the transaction in the stable log, the coordinator will not remember the transaction after it has recovered. If a prepared to commit participant inquires about the transaction's status, the coordinator, not remembering the transaction, will *presume* that the transaction was aborted and responds with an abort message. This special case, in PrN, motivated the design of the two most commonly known 2PC variants that are the first to address in Section 3.

2.1.2 Two-phase commit and multi-level transactions

In the MLTE model, the behaviour of the root coordinator and each leaf participant in the transaction execution tree remains the same as in two-level transactions. The only difference is the behaviour of *cascaded coordinators* (i.e., non-root and non-leaf participants) which behave as leaf participants with respect to their direct ancestors and root coordinators with respect to their direct descendants. Specifically, when a cascaded coordinator receives a prepare-to-commit message, it forwards the message to its direct descendents and waits for their votes. If all descendants have voted 'yes', the cascaded coordinator force-write a prepared log record and then sends a 'yes' vote to its direct ancestor. If any descendant has voted 'no', the cascaded coordinator force-write an abort log record and then, sends an abort decision to the direct descendants that have voted 'yes' and a 'no' vote to its direct ancestor. Once all the required descendants (i.e., direct descendants that have voted 'yes') acknowledge the decision, the cascaded coordinator writes a non-forced end log record and forgets the transaction.

When a cascaded coordinator receives an abort decision (from its direct ancestor), it force-write an abort record and forwards the decision to its direct descendants. Similarly, when a cascaded coordinator receives a commit decision, it force-write a commit record and forwards the decision to its direct descendants. After (locally) complying with the decision that it has received, a cascaded coordinator sends an Ack to its direct ancestor. A cascaded coordinator writes a non-forced end record and forgets the transaction once its (required) direct descendants acknowledge the decision.

2.2 Evaluation issues and metrics

ACPs are commonly evaluated along the lines of three main issues. These issues are as follows (Chrysanthis et al., 1998):

1 *Efficiency during normal processing*: This refers to the cost of an ACP to provide atomicity in the absence of failures. Traditionally, this is measured using three basic metrics. The first metric is *message complexity* which deals with the number of messages that are needed to be exchanged between the systems participating in the execution of a transaction to reach a consistent decision regarding the final status of the transaction. The second metric is *log complexity* which accounts for the frequency at which information needs to be recorded at each participating site in order to achieve resiliency to failures. Typically, log complexity is expressed in terms of the required number of non-forced log records which are written into the log buffer (in main memory) and, more importantly, the number of forced log records which are written onto the stable log (on the disk). The third metric is *time complexity* which corresponds to the required number of rounds or sequential exchanges of messages in order for a decision to be made and propagated to the participants. When each of the previous three metrics is considered individually, it provides a measurement to the amount of overhead over the system's resources. However, the collective amounts of these metrics also affect the performance of the system. This is because some log writes and messages have to be performed in a synchronised sequential manner that delays the release of resources at the participating sites and especially the locks that are placed on data. For this reason, the collective amounts of these metrics affect the level of concurrency of transactions and consequently the overall efficiency of the system.

2 *Resilience to failures*: This refers to the types of failures that an ACP can tolerate and the effects of failures on operational sites. An ACP is considered *non-blocking* if it never requires operational sites to wait (i.e., block) until a failed site has recovered. One such protocol is *three-phase commit* (Skeen, 1981) (that we discuss in Section 4).

3 *Independent recovery*: This refers to the speed of recovery. That is, the time required for a site to recover its database and become operational, accepting new transactions after a system's crash. A site can *independently* recover if it has all the necessary information needed for recovery stored locally (in its log) without requiring any communication with any other site in order to *fully* recover.

Due to the costs associated with PrN during normal transaction processing and its reliability drawbacks in the events of failures, a variety of ACPs have been proposed in the literature. These proposals can be, generally, classified as to enhance either

1 efficiency during normal processing

2 reliability through reduced blocking.

In the next section, we discuss the first class of protocols whereas, in Section 4, we discuss the second class.

3 Efficiency-geared ACPs

The most commonly pronounced 2PC variants are *presumed abort* (PrA) (Mohan et al., 1986) and *presumed commit* (PrC) (Mohan et al., 1986). Both variants reduce the cost of PrN during normal transaction processing, albeit for different final decisions. That is, PrA is designed to reduce the costs associated with aborting transactions whereas PrC is designed to reduce the costs associated with committing transactions.

3.1 Presumed abort

PrA reduces the costs associated with aborting transactions by generalising the special case whereby a coordinator, in PrN, makes an abort presumption about unremembered transactions. That is, PrA is designed such that the coordinator *intentionally* forgets aborting transactions, during normal transaction processing, and uses the abort presumption when replying to the inquiry messages of the participants while keeping its behaviour the same as in PrN for committing transactions. This is in contrast with PrN in which the coordinator uses the abort presumption *only* with undecided transactions and after it has recovered from a system's failure.

Specifically, when a coordinator, in PrA, decides to abort a transaction, it does not force-write the abort decision in its log, as in PrN. Instead, it just sends abort messages to all the participants that have voted 'yes' and discards all information about the transaction from its protocol table. Hence, the coordinator of an aborted transaction does not write any log records for the transaction or needs to wait for Acks from the participants in order to forget the transaction. Since the coordinator of an aborting transaction does not require Acks from the participants in order to be able to forget the transaction, the participants do not have to acknowledge abort decisions. Consequently, the participants do not need to force-write the abort decision onto their logs. Based on this design to PrA, if a participant inquires about a transaction, *at any time*, and the coordinator does not remember the transaction, it means that the transaction must have been aborted. As such, the coordinator replies with an abort message. Thus, as the name implies, if no information is found about a transaction, the transaction is presumed aborted.

3.2 Presumed commit

As opposed to PrA which reduces the costs associated with aborting transactions, PrC is designed, based on the same principle but, to reduce the costs associated with committing transactions. That is, PrC is designed such that missing information about transactions at a coordinator's site is interpreted as commit decisions. However, in PrC, a coordinator has to force-write a commit *initiation* record for each transaction before sending out prepare-to-commit messages to the participants. This record ensures that missing information about a transaction will not be misinterpreted as a commit after a coordinator's site failure without an actual commit decision is made.

To commit a transaction, the coordinator force-write a commit record to logically eliminate the initiation record of the transaction and then sends out the commit decision messages. The coordinator also discards all information pertaining to the transaction from its protocol table. When a participant receives the decision, it writes a non-forced commit record and commits the transaction without having to acknowledge the decision. Based on this design to PrC, if a participant inquires about a transaction, *at any time*, and the coordinator does not remember the transaction, it means that the transaction must have been committed. As such, the coordinator replies with a commit message. Thus, as the name implies, if no information is found about a transaction, the transaction is presumed committed.

To abort a transaction, on the other hand, the coordinator does not write the abort decision in its log. Instead, the coordinator sends out the abort decision and waits for

Acks before discarding all information pertaining to the transaction. When a participant receives the decision, it force-write an abort record and, then, acknowledges the decision, as in PrN. In the case of a coordinator's failure, the initiation record of an interrupted transaction contains all needed information for recovering the transaction.

Both PrC and PrA can be extended to the MLTE model (Mohan et al., 1986) resulting in *multi-level presumed commit* (ML-PrC) and *multi-level presumed abort* (ML-PrA), respectively. The details of these extensions can be found in Al-Houmaily et al. (1997).

Table 1 summarises the costs associated with the three 2PC variants for the commit as well as the abort case assuming a 'yes' vote from each participant: *Total records* is the total number of log records (both forced and non-forced), *Forced records* is the number of forced log writes, *Messages* that appears in the coordinator's column represents the number of messages sent from the coordinator to each participant whereas *Messages* that appears in the participant's column represents the number of reply messages that are sent back to the coordinator from the participant. For simplicity, these costs are calculated for the flat (two-level) transaction execution model.

Table 1 The costs for update transactions in the basic two-phase commit variants

(a) Commit decision

2PC variant	*Coordinator*			*Participant*		
	Total records	*Forced records*	*Messages*	*Total records*	*Forced records*	*Messages*
Basic 2PC (PrN)	2	1	2	2	2	2
Presumed abort	2	1	2	2	2	2
Presumed commit	2	2	2	2	1	1

(b) Abort decision

2PC Variant	*Coordinator*			*Participant*		
	Total records	*Forced records*	*Messages*	*Total records*	*Forced records*	*Messages*
Basic 2PC (PrN)	2	1	2	2	2	2
Presumed abort	0	0	2	2	1	1
Presumed commit	2	1	2	2	2	2

Notes: The highlighted rows indicate that the cost for committing a transaction in PrC is not symmetrical with the cost of aborting a transaction in PrA. This is because of the initiation record that has to be written in a forced manner, in PrC, and that has to be logically eliminated through another forced log write.

3.2.1 Presumed commit variants

Since transactions tend to commit when they finish their execution and reach their commit processing points, the *new PrC* (NPrC) (Lampson and Lomet, 1993) and *rooted PrC* (RPrC) (Al-Houmaily et al., 1997) protocols were both proposed to reduce the log complexity of PrC at the cost of slower recovery in the presence of failures. Specifically, NPrC was proposed to eliminate the forced initiation record from the root coordinator when the flat transaction execution model is used whereas RPrC was proposed to eliminate the forced initiation record from each cascaded coordinator when the MLTE model is used.

Both NPrC and RPrC recognise that the forced initiation record for a transaction, in PrC, serves two purposes at crash recovery time:

1. it identifies the transaction as an aborted transaction (in the absence of an associated commit and end records)
2. it identifies the participants that must acknowledge the abort decision in order for the transaction to be (permanently) forgotten.

Both protocols address these two functions but in different ways.

In NPrC, transactions are assigned monotonically increasing identifiers and the set of available identifiers, at any time, is confined to a fixed range. This range is frequently updated and stored (in a non-forced manner) onto stable storage. After a crash, a coordinator resolves any transaction within its stored range of identifiers as aborted unless it has a commit or end record in its log. Each such transaction requires Acks from the participants but since the participants cannot be determined (due to the elimination of the initiation record that contains their identities), the coordinator cannot request Acks. Consequently, it has to remember these transactions indefinitely. Thus, although NPrC can determine the set of possibly initiated transactions that can be safely considered aborted after a crash, it cannot determine the set of participants that must acknowledge the abort decisions. For this reason, a coordinator stores, for each crash, the set of possibly initiated but unacknowledged transactions onto stable storage which is retained indefinitely (unless a form of a global garbage collection procedure is implemented).

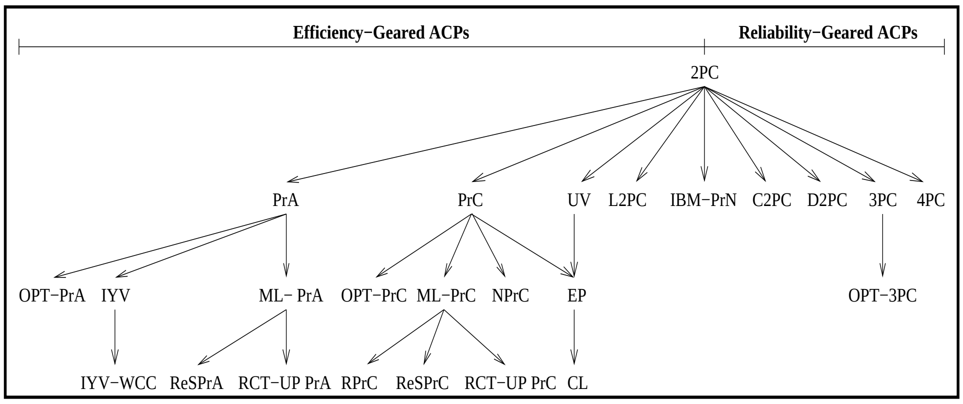
In the MLTE model, the initiation record of the ML-PrC at a cascaded coordinator serves the same two purposes as the ones that it serves at the root coordinator in the flat execution model but for every two adjacent levels. Thus, eliminating this record from cascaded coordinators means revoking their abilities to identify transactions that need to be aborted after a crash and their knowledge about the direct descendants that must acknowledge the abort decisions. For this reason, in RPrC, the initiation record of a root coordinator includes the identities of *all* descendants and not just the immediate ones. Furthermore, for each descendant participant, the initiation record includes the list of the participant's ancestors in the transaction execution tree. In this way, knowledge about transactions that need to be aborted after a crash is *localised* at the root coordinator which also has all the necessary routing descendant information that allows it to request Acks from participants that must acknowledge the decision.

In RPrC, each descendant in a transaction's execution tree includes a list of *all* its ancestors, i.e., superior cascaded coordinator(s), in its forced prepared log record and not just its direct ancestor. This is because if the direct ancestor does not remember the transaction after a crash, it cannot make any presumption about the transaction's final outcome. Thus, the direct ancestor has to consult with its own direct ancestor (about the status of the transaction) and so on until possibly reaching the root coordinator before the status of the transaction can be determined. For this reason, the list of ancestors is included in each inquiry message. This list provides routing information for the ancestors that do not remember the transaction until reaching an ancestor that remembers the transaction and can respond to the inquiry message.

3.3 Other efficiency-geared ACPs

Figure 3 shows some of the significant steps in the evolution of ACPs including, the two most notable 2PC variants, PrA and PrC. In contrast to previous 2PC variants, other variants have been proposed for specific environments. The common characteristic of these protocols is that they exploit the semantics of the communication networks, the database management systems and/or the transactions to enhance the performance of 2PC.

Figure 3 Some significant steps in the evolution of ACPs



Notes: 2PC: two-phase commit [1976], L2PC: linear 2PC [1978], UV: unsolicited-vote [1979], 4PC: four-phase commit [1980], D2PC: decentralised 2PC [1981], 3PC: three-phase commit [1981], C2PC: cooperative 2PC [1981], PrA: presumed abort [1983], ML-PrA: multi-level PrA [1983], PrC: presumed commit [1983], ML-PrC: multi-level PrC [1983], IBM-PrN: IBM's 2PC [1990], EP: early prepare [1990], CL: coordinator log [1990], NPrC: new PrC [1993], IYV: implicit yes-vote [1995], IYV-WCC: IYV with a commit coordinator [1996], RPrC: rooted PrC [1997], ReSPrC: restructured PrC [1997], OPT-PrA: optimistic PrA [1997], OPT-PrC: optimistic PrC [1997], OPT-3PC: optimistic 3PC [1997], RCT-UP PrA: restructured-commit-tree around update-participant PrA [2003], RCT-UP PrC: restructured-commit-tree around update-participant PrC [2003].

3.3.1 Network topology-based ACPs

The *linear* 2PC (L2PC) (Gray, 1978) reduces message complexity at the expense of time complexity compared to 2PC by assuming token-ring like networks. In L2PC, the participants are linearly ordered with the coordinator being the first in the linear order. The coordinator initiates the voting and each participant sends its 'yes' vote to its successor in the linear order. The last participant in the order makes the decision and sends it to its predecessor and so on. In this way, L2PC maintains the same log complexity as 2PC, reduces the message complexity of 2PC from '3*n*' to '2*n*' while increasing the time complexity of 2PC from '3' to '2*n*' rounds, where '*n*' is the number of participants.

In contrast to L2PC, *decentralised* 2PC (D2PC) (Skeen, 1981) reduces time complexity at the expense of message complexity which is 'n^2+n' messages. In D2PC, the interconnecting communication network is assumed to be fully connected and efficiently supports the broadcasting of messages. In D2PC, two rounds of messages are required for each individual participant to make a final decision. During the first round, the coordinator broadcasts its vote (implicitly initiating commit processing) whereas, during the second one, all the participants broadcast their votes. Thus, each participant receives the votes of all the other participants, as well as the coordinator, and thereby, is able to independently conclude the final decision. By reducing the time complexity to two rounds, it becomes less likely for a participant, in D2PL, to be blocked during commit processing in the case of a coordinator's failure.

3.3.2 Application semantics-based ACPs

There are four, transaction type specific, ACPs all of which, when applicable, improve both the message and time complexities of 2PC. In these four protocols, performance enhancement over the basic 2PC is achieved by eliminating the *explicit* voting phase of 2PC, making these protocols look as if they were *one-phase commit* (1PC) protocols. The *unsolicited-vote* protocol (UV) (Stonebraker, 1979) shortens the voting phase of 2PC assuming that each participant knows when it has executed the last operation for a transaction. In this way, a participant sends its vote on its own initiative once it recognises that it has executed the last operation for the transaction. When the coordinator receives the votes of the participants, it proceeds with the decision phase.

The *early prepare* protocol (EP) (Stamos and Cristian, 1993) combines UV with PrC without assuming that a participant can recognise the last operation of a transaction. Every operation is, therefore, treated as if it is the last operation executing at the participant and its Ack is interpreted as a 'yes' vote. This means that a participant has to force-write its log each time it executes an operation so that it can preserve the global atomicity of the transaction after a system crash. Thus, the applicability of EP is limited to systems in which the log file is stored onto an electronic medium whereby forcing a log record is as cheap as writing it into main memory.

In contrast to EP which reduces time and message complexities at the expense of log complexity, the *coordinator log* (CL) (Stamos and Cristian, 1993) and *implicit yes-vote* (IYV) (Al-Houmaily and Chrysanthis, 2000) do not require force writing the log records, at the participants' sites, after the execution of each operation. Instead, they replicate the participants' logs at the coordinators' sites. Hence, reducing log complexity compared to EP at the expense, however, of slower recovery. In CL, a participant does not maintain a local stable log and, therefore, it has to contact *all* coordinators in the system in order to recover after a failure. Moreover, a participant may need to contact the coordinator of a transaction during normal processing to maintain *write-ahead logging* (WAL) or to undo the effects of an aborting transaction. This is because the log of a participant is scattered across the coordinators' sites. In contrast, the log of a participant in IYV is partially replicated across the coordinators' sites. That is, only the *redo* records are replicated at the coordinators' sites while the *undo* records are stored locally. Thus, a participant, in IYV, never communicates with any coordinator to maintain WAL or to undo the effects of aborting transactions. Thus, in IYV, the replicated records are used only to recover a participant after a system's crash. During recovery, a participant needs to contact only those coordinators that are included in its *recovery-coordinators' list* (RCL). This list includes the identities of coordinators with active transactions at the participant's site and is maintained by the participant to limit the number of coordinators that must be contacted after a site failure. Another difference between CL and IYV is that CL is derived from PrC while IYV is derived from PrA.

3.4 Comparison of efficiency-geared ACPs

The basic 2PC and its PrA, PrC and NPrC are general protocols in the sense that they do not make any specific assumptions about the underlying communication network, the data management mechanisms, or the semantics of applications/transactions. Thus, these protocols have a wider applicability range compared to the other variants but at extra log or message overhead. The network-topology based ACPs exploit the connectivity of the underlying communication network to enhance the performance of commit processing. In L2PC, the assumption is that participants can be linearly ordered, using a token-ring like communication network, whereas in D2PC, the assumption is that a participant can directly communicate with each of the other participants, using a fully interconnected communication network.

Application semantics-based ACPs were designed under different assumptions. In UV, it is assumed that each site knows when it has executed the last operation on behalf of a transaction (Stonebraker, 1979). In this way, UV eliminates the explicit voting phase of 2PC and the need to force-write the log records that are generated during the execution of each and every operation prior to acknowledging them. This means that the coordinator either submits to a site all the operations at the same time (which is a form of predeclaration) or indicates to the participant the last operation at the time that the operation is submitted. The former is possible in very special cases while the latter is only possible if each transaction has knowledge about the data distribution in the system and indicates to the coordinator the last operation to be executed at a participant (Al-Houmaily and Chrysanthis, 2000).

Instead of assuming that each site knows when it has executed the last operation on behalf of a transaction, in EP, it is assumed that the cost of force writing a log record is as cheap as writing it into main memory. In this way, a participant prepares itself for commitment after the execution of each operation it performs on behalf of a transaction. Similarly, a coordinator force-write an initiation record each time a new participant joins in the execution of a transaction. This is because EP, which is derived from PrC, requires the identities of the participants to be explicitly recorded at the coordinator's log in a forced initiation log record. Recall that, in PrC, the initiation record allows the coordinator to avoid wrongly presuming commitment of a transaction, after a system's crash, without an actual commit decision is made for the transaction.

To alleviate the drawback of having to force-write multiple initiation records as well as the prepared log records of EP, CL uses *distributed write-ahead logging* (DWAL) and assumes that only the coordinators maintain stable logs. Since a participant might inquire a coordinator about the latest forced log write (i.e., to ensure the WAL), CL might become very costly and less efficient when compared with any of the 2PC variants. Consider the case when a number of long-living transactions execute at a participant without excessive main memory. In this case, the participant might request a transaction's coordinator (explicitly) to force-write its log more than once resulting in a great number of sequential messages. Similarly, rolling back aborted transactions has to be performed completely over the network (since the participants do not maintain their own local logs). This means that when a participant aborts a transaction, it cannot undo the effects of the transaction locally until it communicates with the transaction's coordinator and receives the undo log records pertaining to the transaction, which is a significant overhead. In addition, undoing the effects of aborted transactions *logically*, using ARIES (Mohan et al., 1992), the proposed recovery scheme for CL, require the execution of further operations that has to be executed and (remotely) logged. This represents an added overhead to the cost of CL.

Although both CL and IYV do not make any assumptions about the structure of transactions, which is in contrast to UV, IYV also does not need to force-write initiation records as it is derived from PrA. Furthermore, WAL is maintained locally in IYV without the need to communicate with any coordinator as each participant maintains its own (local) log. Consequently, aborted transactions are handled locally without any communication with the coordinators (i.e., the undo log records do not have to be requested from the coordinators). Another significant difference between IYV and CL is the case of a coordinator's recovery after a failure. A coordinator in IYV can recover independently without communicating with any participant. In contrast, a recovering coordinator, in CL, has to communicate with all possible participants in the system in order to determine the set of unknown transactions in order to abort them instead of presuming their commitment since CL is a descendant from PrC protocol (Stamos and Cristian, 1993). For a recovering participant in CL, the participant has to wait until it receives all the log records from the coordinators and until all active transactions have been decided upon. In IYV, however, only the coordinators in the RCL needs to be contacted by a recovering participant and using the 'still active' message received from a coordinator, that is in the list, the participant can recover its state up to the point prior to its failure and resume its normal processing without having to wait until all active transactions have been decided upon, allowing long-living transactions to forward recover and resume their execution.

Although each of the above semantics-based ACPs enhances the performance of commit processing under specific assumptions, all of them require the termination of transactions before reaching their commit processing stages. That is, once a transaction reaches its commit processing stage, it is not allowed to execute any further operations or even acquire any more locks on data. As such, these protocols suffer from a common limitation which is their inability to support the option of *deferred consistency constraints* that is currently part of the SQL standards. Circumventing this limitation is discussed in Section 5.2.2.

3.5 Performance of efficiency-Geared ACPs

This section analytically evaluates the performance of efficiency-geared ACPs along the lines of the three performance issues presented in Section 2.2.

3.5.1 Efficiency during normal processing

For the evaluation of *efficiency during normal processing*, the basic performance metrics (i.e., log, message and time complexities) are divided further into the following metrics:

- *Log overhead* corresponds to *log complexity* and is divided into '*Total*' number of log records and the portion of total log records that needs to be '*Forced*' written.

- *Log delays* represents the number of '*Sequential*' forced log writes that are required up to the point that the commit/abort decision is made.
- *Message overhead* corresponds to *message complexity* and is divided into '*Total*' number of messages, number of messages required for a '*Decision*' to be made and propagated to the participants, and the portion of total messages that are due to '*DWAL*' (in CL).
- *Message delays* corresponds to *time complexity* and is divided into the number of required sequential messages up to the '*Decision*' point and the number of required sequential messages for the release of all the '*Locks*'.

Based on the above divisions to the basic performance metrics, Table 2 compares the different protocols under the worst case scenario. This scenario is expected to reflect the average behaviour of transactions and the distributed environment because the assumptions made in this scenario are more realistic than the assumptions made in the best case scenario that we discuss next. In the table, '*n*' represents the number of participants that executed a transaction and '*d*' represents the number of data items that have been accessed by the transaction. This scenario is based on the following assumptions (Al-Houmaily and Chrysanthis, 2000):

- A transaction has more than one write operation at each participant it accesses (i.e., $d > n$).
- Transactions execute serially (i.e., an operation is submitted by a transaction only after the previous operation has been executed and acknowledged).
- The participants are not known at the beginning of transactions.
- The participants in a transaction execution do not have excessive main memories and each operation generates a single log record. This means that each and every log record that is generated due to the execution of an operation has to be forced written onto the stable log as a worst case scenario. Note that we do not include the number of forced log writes that are due to the operations and which are the same in all the protocols except for EP where the log records have to be forced written all the time. In CL, on the other hand, operations add extra overhead because each force-write is explicitly associated with two messages due to DWAL.

For example, Table 2 shows that the cost of committing a transaction, in 2PC, is *exactly* the same as the cost of aborting a transaction. For either case, the '*Total*' *log overhead* is '*2n+2*', out of which '*2n+1*' are '*Forced*' log records. The '*Sequential*' *log delays* for a decision to be made, in 2PC, is '*2*': one at the participant(s) and one at the coordinator. The '*Total*' *message overhead* is '*4n*', out of which '*3n*' are required for a '*Decision*' to be made and propagated to all the participants. The '*DWAL*' *message overhead* is '*0*' for 2PC as only CL uses DWAL among the evaluated protocols. The sequential '*Decision*' *message delays* is '*2*' as 2PC requires two sequential messages for a decision to be made whereas the '*Locks*' *message delays* is '3' as 2PC requires three sequential messages for a decision to reach a participant and releases the locks held by a transaction.

Table 2 Protocols' costs for the flat execution model assuming *worst* case scenario

(a) Commit decision

Metric / *ACP*	*Log overhead*		*Log delays*	*Message overhead*			*Message delays*	
	Total	*Forced*	*Sequential*	*Total*	*Decision*	*DWAL*	*Decision*	*Locks*
2PC	$2n+2$	$2n+1$	2	$4n$	$3n$	0	2	3
PrC	$2n+2$	$n+2$	3	$3n$	$3n$	0	2	3
PrA	$2n+2$	$2n+1$	2	$4n$	$3n$	0	2	3
NPrC	$2n+1$	$n+1$	2	$3n$	$3n$	0	2	3
L2PC	$3n+1$	$2n+1$	n	$3n$	$2n$	0	n	$2n$
D2PC	$3(n+1)$	$2(n+1)$	3	$2(n^2+n)$	n^2+n	0	2	2
UV	$2n+2$	$2n+1$	2	$2n$	n	0	0	1
EP	$d+2n+1$	$d+n+1$	$d+n+1$	n	n	0	0	1
CL	1	1	1	$2d+n$	$2d+n$	$2d$	$2d$	$2d+1$
IYV	$n+2$	1	1	$2n$	n	0	0	1

(b) Abort decision

Metric / *ACP*	*Log overhead*		*Log delays*	*Message overhead*			*Message delays*	
	Total	*Forced*	*Sequential*	*Total*	*Decision*	*DWAL*	*Decision*	*Locks*
2PC	$2n+2$	$2n+1$	2	$4n$	$3n$	0	2	3
PrC	$2n+2$	$2n+1$	2	$4n$	$3n$	0	2	3
PrA	$2n$	n	1	$3n$	$3n$	0	2	3
NPrC	$2n+1$	$2n$	1	$4n$	$3n$	0	2	3
L2PC	$3n+1$	$2n+1$	n	$3n$	$2n$	0	n	$2n$
D2PC	$3(n+1)$	$2(n+1)$	3	$2(n^2+n)$	n^2+n	0	2	2
UV	$2n+2$	$2n+1$	2	$2n$	n	0	0	1
EP	$d+2n+1$	$d+2n$	$d+n$	$2n$	n	0	0	1
CL	1	0	0	$4d+n$	$2d+n$	$4d$	$2d$	$4d+1$
IYV	n	0	0	n	n	0	0	1

It is clear from Table 2 that IYV and CL outperform all the other protocols with respect to the number of '*Forced*' *log overhead* as well as the number of '*Sequential*' *log delays*. For the commit case, the two protocols require only one log force-write whereas for the abort case neither IYV nor CL force-write any log records. In this respect, EP is the most expensive of all protocols.

CL becomes more expensive than IYV when *Message overhead* and *Message delays* are considered. Due to DWAL, CL requires two explicit sequential messages to be exchanged between a participant and the coordinator of a transaction for each operation executed by the participant for the commit case (thus, the '*2d*' in '*DWAL*' *message overhead*). For the abort case, four messages are needed to be exchanged between the participant and the coordinator of an aborted transaction. This is because undoing an operation using the recovery scheme of CL, ARIES, is another operation that has to be executed and logged. Since CL uses DWAL, undoing an operation requires two more explicit messages to be exchanged between the coordinator and the participant in the worst case scenario. Note that because of its dependency on the number of data operations, CL can potentially involve more messages to commit or abort a transaction than any of the other protocols in the case of long-transactions. On the other hand, with respect to *Message overhead* and *Message delays*, IYV and EP perform better than the other protocols. For the commit case, EP incurs the least number of '*Total*' *message overhead* whereas for the abort case, IYV incurs the least.

As the performance of EP and CL is highly dependant on the behaviour of transactions, which is unlike the other protocols, Table 3 contains the performance evaluation of these two protocols based on the ideal case scenario. The table also repeats the performance evaluation of UV and IYV just to simplify comparisons among application semantics-based ACPs under the ideal case scenario. This scenario is based on the following assumptions (Al-Houmaily and Chrysanthis, 2000):

- A transaction performs at most one write operation per each site it accesses (i.e., $n = d$).
- The operations of a transaction execute in parallel.
- The participants are known at the beginning of transactions.
- The participants have excessive main memory.

In the ideal case scenario, both CL and IYV have the same costs as far as the number of '*Forced*' *log overhead* and '*Sequential*' *log delays* are concerned with CL better than IYV in the cost of '*Total*' *log overhead*. In this respect, CL and IYV dominate all the other protocols with CL dominating IYV by '*n*' in the '*Total*' *log overhead*. For *Message overhead* and *Message delays*, the four protocols require the same costs, for both the commit as well as the abort case, with the exception in the '*Total*' *message overhead* that depends on the presumption used in the base protocol from which the evaluated protocol is derived. For example, IYV requires a total of '2*n*' messages for the commit case and '*n*' messages for the abort case because it is derived from PrA in which only commit decisions need to be acknowledged. In this respect, the cost of CL is exactly the opposite compared to IYV: '*n*' for the commit case and '*2n*' for the abort case.

Table 3 Protocols' costs for the flat execution model assuming *ideal* case scenario

(a) Commit decision

ACP \ *Metric*	*Log overhead*		*Log delays*	*Message overhead*			*Message delays*	
	Total	*Forced*	*Sequential*	*Total*	*Decision*	*DWAL*	*Decision*	*Locks*
UV	2*n*+2	2*n*+1	2	2*n*	*n*	0	0	1
EP	2*n*+2	*n*+2	3	*n*	*n*	0	0	1
CL	1	1	1	*n*	*n*	0	0	1
IYV	*n*+2	1	1	2*n*	*n*	0	0	1

(b) Abort decision

ACP \ *Metric*	*Log overhead*		*Log delays*	*Message overhead*			*Message delays*	
	Total	*Forced*	*Sequential*	*Total*	*Decision*	*DWAL*	*Decision*	*Locks*
UV	2*n*+2	2*n*+1	2	2*n*	*n*	0	0	1
EP	2*n*+2	2*n*+1	2	2*n*	*n*	0	0	1
CL	1	0	0	2*n*	*n*	0	0	1
IYV	*n*	0	0	*n*	*n*	0	0	1

Comparing the two scenarios, we conclude that the cost associated with EP is highly dependent on the number of operations submitted by transactions while CL is also dependent on the behaviour of the system. This makes EP and CL completely inefficient when used in distributed database systems with long-living transactions in which transactions typically execute a large number of operations at a small number of sites (i.e., $d > n$).

3.5.2 Resilience to failures

All the above protocols can tolerate both site and communication failures and, at the same time, they are all susceptible to blocking in the case of such failures although the size of the window of vulnerability and the amount of blocked data is different in the above protocols. More specifically, a participant in EP, CL and IYV is susceptible to blocking after acknowledging the execution of each and every operation of a transaction. This is in contrast to the other protocols in which a participant is susceptible to blocking only after a transaction has executed all it operations and across all participating sites. Thus, the window of vulnerability is larger in EP, CL and IYV than the other protocols. On the other hand, the amount of blocked data in EP, CL and IYV depends on the timing at which a failure occurs during the execution of a transaction. That is, in EP, CL and IYV, the amount of blocked data depends on the progress of the transaction (i.e., the number of operations that the transaction has executed) before the occurrence of the failure. This amount of data does not necessarily correspond to the total number of data objects that are to be blocked had the transaction executed to its last operation. In contrast to this incremental blocking of data at a participant, in the other protocols, all data is blocked at the participant at the same time or no data is blocked at all. That is, during the execution phase of a transaction, no data objects need to be blocked if a failure occurs but once the transaction enters it's prepared to commit state, all data objects need to be blocked.

3.5.3 Independent recovery

As stated in Section 2.2, independent recovery means the ability of a site to recover the state of its database and become fully operational again after a system crash, accepting new transactions for execution, without the need to communicate with any other site to complete its recovery process. Thus, independent recovery is not be confused with the issue of blocking as blocking affects a site during its normal operation (or after it has become operational after recovering from a crash) due to its inability to communicate with the coordinator(s) of undecided transactions and resolve their status. Based on this definition to independent recovery, all the protocols discussed thus far, with the exception of CL and IYV, allow for independent recovery. This is because, in these protocols, all information needed for recovery can be found locally in the stable log of the recovering site whether it is a coordinator or a participant.

On the other hand, both CL and IYV are characterised by their lack of independent recovery. This is because some required information for recovery, in both protocols, is stored at remote sites and the recovery process after a system crash has to be suspended until this information becomes available locally. Because of the absence of needed information for recovery, in CL, neither a coordinator nor a participant can recover independently whereas, in IYV, only a participant cannot recover independently. In CL, a recovering coordinator has to determine the set of unknown transactions and resolve their status. Recall that CL is based on the same presumption as PrC and, at the same time, it eliminates the forced initiation record that is used in PrC. Thus, the ability of a coordinator, in CL, to unilaterally and precisely determine the set of active transactions at its site prior to a system crash is curtailed. For this reason, the coordinator needs to communicate with all possible participants to determine this set of transactions and abort them before it can finish its recovery process. This is to avoid latter presuming wrongly their commitment. In contrast to CL and being based on PrA, a coordinator, in IYV, does not need to communicate with any participant to determine such a set of transactions. If a transaction was active prior to a coordinator failure and a participant inquires about the transaction after the coordinator has recovered, the coordinator will correctly presume that the transaction was aborted and responds with an abort message.

For a recovering participant in both CL and IYV, the participant has to wait until it receives all the required log records from the coordinators. However, as the participants in CL do not maintain local logs, a recovering participant has to wait until all coordinators reply to its ‘recovery’ message whereas, in IYV, a recovering participant needs to wait only for the replies of those coordinators that are included in its RCL. Not only that, but a recovering participant, in IYV, can proceed with the undo phase of its recovery process while waiting for the reply messages to arrive from the required coordinators, possibly masking any communication delays required for the delivery of the required recovery information from the required coordinators.

From the above, it is clear that the efficiency-geared ACPs exhibit variant degrees of independent recovery with CL exhibiting the lowest degree among the other evaluated protocols.

4 Reliability-geared ACPs

In 2PC, a participant is *blocked* if it fails to communicate with the coordinator of a transaction while in a prepared to commit state. Blocking means that the participant cannot determine the final status of the transaction in the presence of a failure, rendering all resources held by the prepared to commit transaction at its site unusable by any other transaction until the final status of the transaction is resolved, i.e., the transaction is either committed or aborted.

Blocking is inevitable in 2PC and it may occur either because of

1 a communication (link) failure

2 a coordinator’s system crash.

These two types of failures lead to blocking, in 2PC, even under the assumption that all participants remain operational and can communicate collaboratively to resolve the status of the prepared to commit transaction. For example, if a coordinator fails after all participating sites in a transaction’s execution have entered their prepared to commit states, the participants (collectively) can neither commit nor abort the transaction. This is because the operational participants cannot be sure whether the coordinator had received all their votes and made a commit final decision just before it failed or it had received only some votes and did not have the chance to make the final decision before its failure and the transaction will be aborted by the coordinator when it recovers. Thus, the participants are blocked until the coordinator recovers.

The negative impact of blocking on the

1 overall system performance

2 availability of critical data on other transactions motivated the design of *non-blocking* ACPs.

Three-phase commit (3PC) was one of the first attempts to resolve the blocking aspects of 2PC (Skeen, 1981). It alleviates the *blocking* aspect of 2PC in the events of *site* failures. That is, 3PC never requires operational sites to wait (i.e., block) until a failed site has recovered. The main purpose of the protocol is to allow operational sites to continue transaction processing and reach agreement about the final status of transactions in spite of the presence of site failures. 3PC can tolerate any number of site failures (except for total sites' failures), assuming a highly reliable network (i.e., a network that never causes operational sites to be partitioned into more than one set of communicating sites, implying a network that never fails) (Skeen and Stonebraker, 1983).

4.1 The three-phase commit protocol

In 3PC, an *extra (buffering) phase* is inserted between the two phases of 2PC to capture all the participants' intentions to commit, as shown in Figure 4. The basic idea behind the insertion of the buffering phase is to place the two (possibly) reachable final states (i.e., the commit and the abort states) for a transaction apart from each other such that they cannot be reached from the *same* state. That is, if a final state can be reached from the current state of a site, then, the reachable final state can be either the commit or the abort state but not both. In 2PC, when a participant is in a prepared to commit state, both final states can be reached. Hence, if the coordinator fails, the participant cannot determine the final state for the transaction without any possible conflict in its decision with the coordinator. For this reason, the protocol is blocking. On the contrary and as shown in Figure 4, the commit final state for a site (whether it is the coordinator or a participant), in 3PC, cannot be reached from the same state as the abort final state. In the former case, the commit state can be reached from the *pre-commit* state whereas, in the latter case, the abort state can be reached from the *prepared* state.

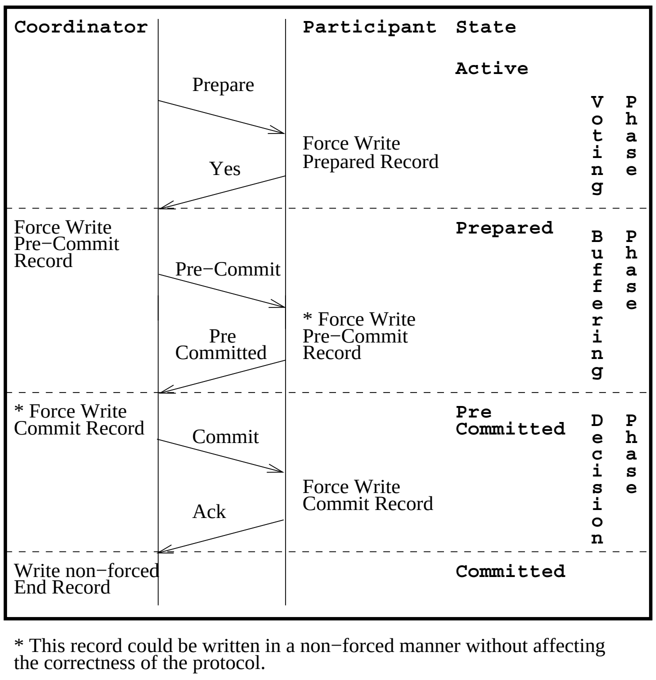
When a site is in a pre-commit state, it means that each of the other sites is at least in a prepared to commit state[1]. Thus, the pre-commit state is called a *committable state* since it implies that all participants have voted 'yes' and the coordinator agreed on the commitment of the transaction. In 3PC, a *non-committable* state, i.e., a state that does not imply that all the participants have voted 'yes', is not placed adjacent to a commit state. This is not the case in 2PC as the prepared state, which non-committable, is placed adjacent to a commit state.

The insertion of the buffering state makes the structure of 3PC to satisfy the two necessary and sufficient conditions for the construction of synchronous non-blocking ACPs within one state transaction, i.e., a structure where neither the coordinator nor any participant leads each other by more than one state transition during its execution of the protocol. That is, 3PC is synchronous within one state transaction that

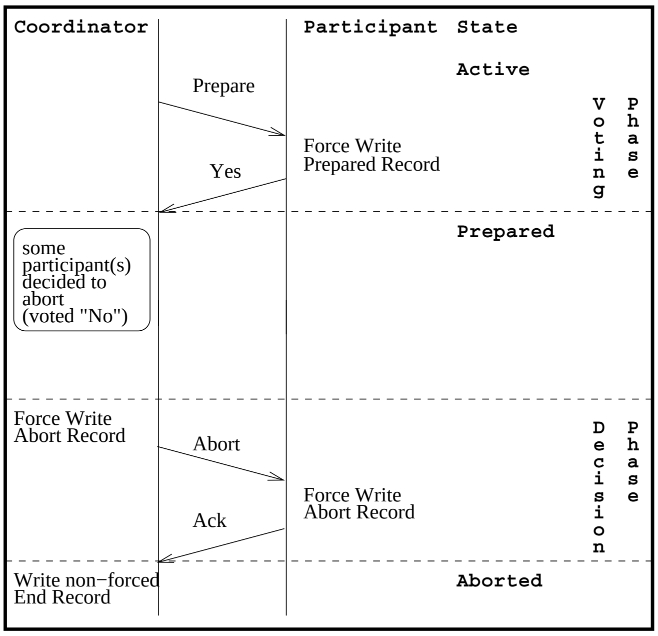
1 does not contain a state that is adjacent to both a commit and an abort state

2 it does not contain a non-committable state that is adjacent to a commit state.

Figure 4 The three-phase commit protocol (a) commit case and (b) abort case


Coordinator
Participant
State
Active
Prepare
Force Write Prepared Record
Yes
Voting Phase
Force Write Pre−Commit Record
Prepared
Pre−Commit
* Force Write Pre−Commit Record
Pre Committed
Buffering Phase
* Force Write Commit Record
Pre Committed
Commit
Force Write Commit Record
Ack
Decision Phase
Write non−forced End Record
Committed
* This record could be written in a non−forced manner without affecting the correctness of the protocol.

(a)



(b)

Based on the above, if the coordinator of a transaction fails at any point during the execution of the protocol, the operational participants can collectively and deterministically decide the final status of the transaction. The decision is commit if any of the participants is in *at least* a pre-commit state (because it is not possible for the coordinator to have decided to abort). Otherwise, the decision is abort (because it could be possible for the coordinator to have decided to abort but not to commit). To reach an agreement on the final status of a transaction, there is a need for a termination protocol which is invoked when the coordinator fails.

4.1.1 Recovery in three-phase commit

When a participant times out while waiting for a message from the coordinator, it means that the coordinator must have failed (or it is perceived as a coordinator's failure). In this case, the participant initiates an election protocol to determine a *new* coordinator. One way to determine the new coordinator is based on sites' identification numbers such that the participant with the highest (or the lowest) number becomes the new coordinator. Once a new coordinator is elected, the participants exchange status information about the transaction. If the new coordinator finds the transaction in at least a pre-commit state at any participant, it commits (in its local log) the transaction; otherwise, it aborts the transaction. Then, this new coordinator proceeds to complete the 3PC for the transaction in all the other participants. If the new coordinator fails, the election process is repeated again.

When a participant starts recovering from a failure, it needs to determine the status of each prepared or pre-committed transaction as recorded in its log. Notice that a recovering participant cannot commit a transaction even if the participant is in a pre-commit state with respect to the transaction. This is because the operational sites might have decided to abort the transaction after the participant had failed if none of them was in a pre-commit state. In this case, the participant must ask the other sites about the final status of the transaction.

Another important issue related to failures in 3PC is total sites' failure where there is a need to determine the last participant to have failed. This is because such participant is the only one which can decide the status of the transaction for the other participants. Determining the last participant that has failed could be implemented by maintaining an 'UP' list at each participant. This list contains the identities of operational participants as seen by the participant that is maintaining the list and is stored in a non-forced manner onto the stable log of the participant. Thus, the 'UP' lists allow a set of participants to determine, upon their recovery from the total failure, whether they contain among themselves the last participant to have failed, reducing the number of participants that needs to recover before the transaction status can be resolved. Alternatively, all participants should recover and become operational again before the status of the transaction can be resolved.

4.2 Other reliability-geared ACPs

As discussed above, blocking occurs in 2PC when the coordinator of a transaction crashes while the transaction is in its *prepared to commit* state at a participating site. In such a case the participant is blocked until the coordinator of the transaction recovers. In general, all ACPs are susceptible to blocking (Skeen and Stonebraker, 1983). They just differ in the size of the window during which a site might be blocked and the type of failures that cause their blocking. Several ACPs have been designed to eliminate some of the blocking aspects of 2PC, besides 3PC, by adding extra coordination messages and forced log writes. These protocols can be classified into whether they preserve the prepared to commit state or allow *unilateral or heuristic* decisions in the presence of unbearable delays. Examples of the former, other than 3PC, are *cooperative* 2PC (C2PC) (LeLann, 1981) and *four-phase commit* (4PC) (Hammer and Shipman, 1980), whereas *IBM presumed nothing* (IBM-PrN) (Samaras et al., 1995) is an example of the latter.

C2PC reduces the *likelihood* of blocking in case of a coordinator's failure. In C2PC, the identities of all participants are included in the prepare-to-commit message so that each participant becomes aware of the other participants. In the case of a coordinator's or a communication's failure, a participant does not block waiting until it re-establishes communication with the coordinator. Instead, it inquires the other operational participants in the transaction's execution about the final decision and if any of them has already received the final decision prior to the failure, it informs the inquiring participant accordingly.

Even though C2PC reduces the likelihood of blocking in the case of a coordinator's site failure, it is still subject to blocking in the event that the coordinator fails. This occurs when the coordinator fails while all participants are in their prepared to commit states. In contrast to C2PC, 4PC eliminates this type of blocking but in a manner that is different from 3PC. Specifically, 4PC increases the number of sites that might have status information about a transaction in the case of a coordinator's failure rather than increasing the number of states (though the insertion of the extra buffering state). This is accomplished through the initiation of 2PC with a number of back up sites that are linearly ordered. The back up sites do not participate in the transaction execution per se but they increase the number of sites that might have status information about the transaction in the case of a coordinator's failure. Once the back up sites have acknowledged the commitment of the transaction, the coordinator initiates 2PC with the rest of the participants. Thus, in the case of a coordinator's failure, the back up site with the least identifier in the order that is still operational takes over as the new coordinator and commits the transaction as in 2PC.

The IBM-PrN is a 2PC variant that allows blocked participants to unilaterally commit or abort a transaction and detects atomicity violations due to conflicting heuristic decisions. In the event of atomicity violations, it reports any damage on transactions and data, simplifying the task of identifying problems that must be fixed. The *generalised presumed abort* (GPrA) (Mohan et al., 1993) is another IBM protocol that behaves like IBM-PrN when complete confidence in the final outcome and recognition of heuristic decisions is required and behaves like PrA during normal processing. Other efforts to enhance commit protocols with heuristic decision processing resulted in the *allow-heuristics presumed nothing* commit protocol (Samaras and Nikolopoulos, 1995).

5 Integration of ACPs

ACPs are incompatible with each other and therefore need to be made to interoperate in order to be integrated in (heterogeneous) multi-database systems and the internet. Thus, the continued research for more pragmatic or efficient ACPs has expanded to include the investigation of integrated ACPs. Early efforts in this direction include the harmony prototype system that integrates *centralised* participants that use either centralised (asymmetric) or decentralised (symmetric) ACPs (Pu et al., 1991), and the integration of *distributed* participants that use symmetric or asymmetric ACPs (Tal and Alonso, 1994). Symmetric participants are those participants that use ACPs in which their roles, in commit processing, are the same as the role of the coordinator, such as D2PC; whereas asymmetric participants are those participants that use ACPs in which their roles, in commit processing, are different from the role of the coordinator, such as PrN. On the other hand, a centralised participant represents a standalone participant that implements an ACP whereas a distributed participant represents a set of sites that implement the same ACP.

5.1 Sources of incompatibilities

The above two research works concentrated on resolving the incompatibility issues arising out of the semantics of messages. These issues include both the meanings of messages as well as their existence. For example, in D2PC, the prepare-to-commit message of the coordinator has a dual role:

1 it tells each participant that the transaction has finished its execution, thereby initiating the voting phase, and at the same time

2 it reveals, to each participant, the coordinator's vote.

Thus, the *meaning* of the prepare message *is different* in D2PC than its meaning in, for example, PrN where it initiates the voting phase but does not reveal the coordinator's vote. On the hand, a participant in D2PC expects to receive the votes of all the other participants in the execution of the transaction which are never generated by, for example, participants that adopt PrN. Thus, such a D2PC participant will encounter missing messages when integrated with 2PC participants while a 2PC participant will encounter extra (out-of-protocol) messages when integrated with D2PC participants.

As the above examples (briefly) demonstrate, ACPs are incompatible since that they cannot be used (directly) in the same environment without conflicts. This is true even for the simplest and most closely related variants such as PrN, PrA and PrC. Detailed analysis to the sources of incompatibilities among ACPs shows that it could be due to

1 the semantics of the coordination messages (which include both their meanings as well as their existence, as mentioned above)

2 the presumptions about the outcome of terminated (and forgotten) transactions in case of failures (Al-Houmaily, 2008).

Problems arising out of presumption incompatibilities could lead to either

1 impractical integration of ACPs

2 violations to the global atomicity of transactions (Al-Houmaily and Chrysanthis, 1996a; Al-Houmaily and Chrysanthis, 1999; Al-Houmaily, 2008).

Impractical integration means that the outcome of a terminated transaction might have to be remembered forever, curtailing the system operation on the long run. On the other hand, atomicity violations mean that a transaction might end up committing at some sites and aborting at others due to a system failure.

5.2 Resolving presumption incompatibility

The *presumed any* (PrAny) protocol was the first protocol to recognise the problem of presumption incompatibilities and resolving it in a practical manner (Al-Houmaily and Chrysanthis, 1996a; Al-Houmaily and Chrysanthis, 1999). PrAny interoperates PrN, PrA and PrC in the context of multi-database systems, a special case of heterogeneous distributed databases in which each database site is pre-existing, supporting its own local applications and users, and has some degree of local autonomy. Specifically, PrAny was used to demonstrate the difficulties that arise when one attempts to interoperate participants that adopt ACPs with different presumptions about the outcome of terminated transactions in the same environment and, more importantly, to introduce the *operational correctness criterion* and the notion of *safe state*. Operational correctness means that *all* participating sites should be able, not only to reach an agreement but also, to (eventually) forget the outcome of terminated transactions. On the other hand, the safe state means that, for any operationally correct ACP, the coordinator should be able to reach a state in which it can reply to the inquiry messages of the participants, in a consistent manner, without having to remember the outcome of terminated transactions forever. Formal definitions of operational correctness and safe state can be found in (Al-Houmaily and Chrysanthis, 1999).

5.2.1 The presumed any protocol

PrAny interoperates PrN, PrA and PrC in accordance to the operational correctness criterion by

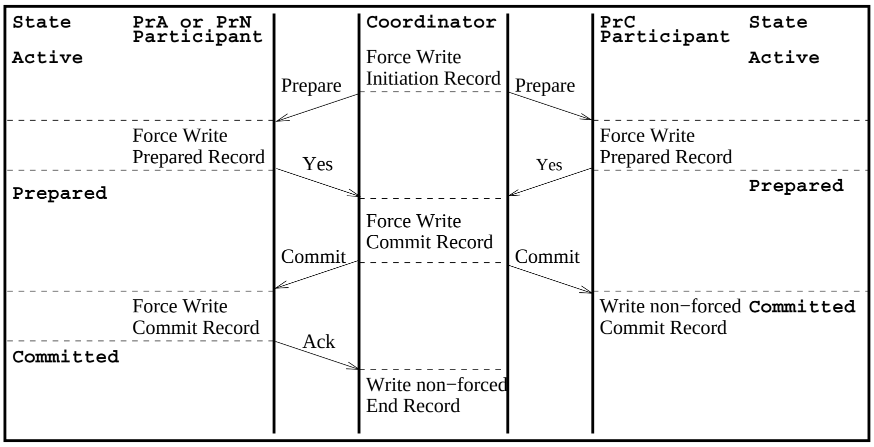
1 ‘talking’ the language that each of the three protocols understands with respect to messages

2 synchronising the timing at which it forgets the outcome of terminated transactions.

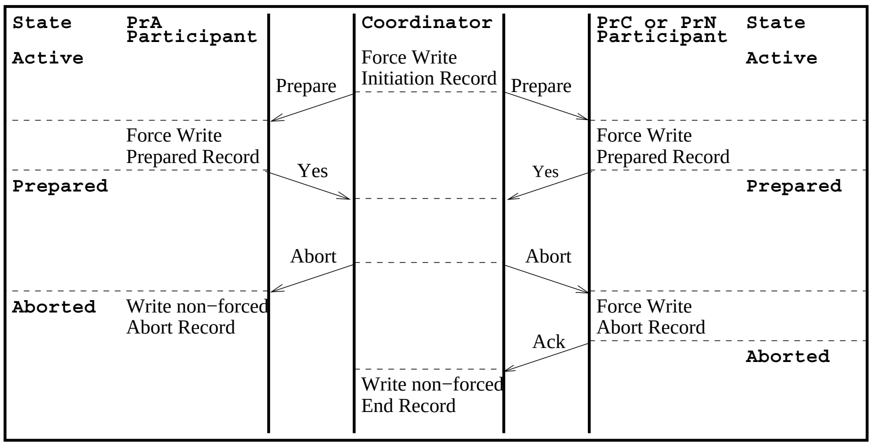
By talking the language of the three protocols, PrAny enables the different 2PC participants to reach agreement whereas, by synchronising the timing at which a coordinator forgets a transaction, PrAny allows a coordinator of a transaction to reach a safe state with respect to the transaction.

As shown in Figure 5, if a coordinator talks the language of the three 2PC variants and knows which variant is used by which participant, it should forget a committed transaction once all PrA and PrN participants acknowledge the commit decision, and should forget an aborted transaction once all PrC and PrN acknowledge the abort decision. This is because only commit decisions are acknowledged in PrA [see the PrA participant on the left hand side of Figure 5(a)] whereas, in PrC, only abort decisions are acknowledged [see the PrC participant on the right hand side of Figure 5(b)]. Recall that a coordinator needs to, eventually, forget the outcome of terminated transactions to comply with the operational correctness criterion. However, a coordinator’s protocol which only talks the language of the three variants is not sufficient since it might violate the atomicity of transactions in case of failures. This is because the coordinator cannot adopt a single presumption about forgotten transactions that always matches their actual final outcomes before they had been forgotten. For example, if the coordinator adopts, for recovering purposes, the abort presumption, it will respond with an abort message to a recovering PrC participant that inquires about a forgotten committed transaction [Figure 5(a)]. Similarly, if the coordinator adopts the commit presumption, it will respond with a commit message to a recovering PrA participant that inquires about a forgotten aborted transaction [Figure 5(b)]. Thus, such a protocol might violate transactions' atomicity. Alternatively, the coordinator should remember the outcome of transactions forever, violating operational correctness.

Figure 5 The presumed any protocol (a) commit case and (b) abort case


State
PrA or PrN Participant
Coordinator
PrC Participant
State
Active
Force Write Initiation Record
Active
Prepare
Prepare
Force Write Prepared Record
Yes
Yes
Force Write Prepared Record
Prepared
Prepared
Force Write Commit Record
Commit
Commit
Force Write Commit Record
Write non−forced Commit Record
Committed
Ack
Committed
Write non−forced End Record

(a)



(b)

PrAny remedies the above dilemma by having the coordinator adopt the presumption of the protocol used by the inquiring participant rather than adopting a single coordinator's presumption. That is, if a participant inquires about a forgotten committed transaction, the participant has to be a PrC participant. This is because only PrC participants do not acknowledge commit decisions. Thus, the coordinator will reply with a commit message in accordance with PrC adopted by the participant. On the other hand, if a participant inquires about a forgotten aborted transaction, the participant has to be a PrA participant. This is because only PrA participants do not acknowledge abort decisions. Thus, the coordinator will reply with an abort message in accordance with PrA adopted by the participant. Knowledge about the used protocols by the participants is recorded statically, in a *participants*' *commit protocol* (PCP) table, at the coordinator's site.

The above presumption strategy in PrAny allows participants that adopt ACPs with contradicting presumption to participate in the execution of the same transaction without causing atomicity violations to transactions or affecting the practicality of the protocol (i.e., violating operational correctness). Although knowledge about the used protocols by the participants has to be recorded statically, the protocol allows for the switching from one protocol to another on a per transaction basis and on a per participant basis (Al-Houmaily and Chrysanthis, 1996a). This is in order to trade-in coordination messages for forced log writes, enhancing the overall commit processing performance when the cost of messages are more expensive than the cost of forced log writes. Recoding the used protocol by the coordinators, in PrAny, is a necessity since the protocol was designed in the context of multi-database systems where each participant is characterised by its autonomy, implying that it might be using a protocol that differs from the other participants in the environment without the possibility of changing (or even modifying) its used protocol.

Systematic analyses to the sources of incompatibilities when interoperating 2PC protocols with 1PC protocols, such as IYV, led to the design of the *integrated two-phase commit* (I-2PC) protocol (Al-Houmaily, 2008). That is, besides identifying the sources of incompatibilities when one attempts to interoperate 2PC with 1PC protocols, I-2PC generalises PrAny by incorporating IYV, which is a 1PC protocol, besides PrN, PrA and PrC.

5.2.2 Other incompatibility resolving protocols

The motivation behind the design of PrAny and I-2PC was for a pragmatic reason. That is, achieving atomicity of transactions, according to the operational correctness criterion, in spite of the heterogeneity of the used ACPs by the participants. Thus, performance was not a main issue behind the design of PrAny and I-2PC, although different optimisations for PrAny are still possible (Al-Houmaily and Chrysanthis, 1996a) as mentioned above. In contrast, the main motivation behind the design of *presumed-either* (PE) (Attaluri and Salem, 2002), *dynamic presumption two-phase commit* (DPr-2PC) (Yu and Pu, 2007), *one-two phase commit* (1-2PC) (Al-Houmaily and Chrysanthis, 2004a, 2004b) and *adaptive participant's presumption protocol* (AP^3) (Al-Houmaily, 2005) was for performance reasons. That is, enhancing the performance of atomic commit processing through the use of adaptive integrated ACPs rather than using only a single ACP at all times and with all transactions. For this reason, these protocols were designed implicitly assuming *homogenous* distributed database systems in which all sites deploy the same (integrated) ACP.

The design of PE is based on a note that was mentioned in (Mohan et al., 1986) and discussed atomic commit processing in System R*, one of the earliest relational prototype systems that was built at IBM research laboratories. It stated that both PrA and PrC might be used side-by-side in the same environment with the protocol choice being made on a *per transaction* basis and at the beginning of the commit processing stage of the transaction. That is, if a transaction is expected to be aborted during commit processing rather than being committed, PrA is used. Otherwise, PrC is used. In this way, the overall system performance is enhanced since it is cheaper to use PrA with aborting transactions and cheaper to use PrC with committing transactions, as previously discussed in Section 3. Such a protocol could be called *System's R* presumed either* (R^* PE).

Motivated by the fact that transactions tend to commit when they reach their commit processing stage and that forcing the initiation record in PrC represents its main performance efficiency obstacle for committing transactions when compared with the performance of PrA for aborting transactions (as highlighted previously in Table 1), PE was designed to eliminate the *forcing* of the initiation record of PrC *whenever possible*. That is, PE does not actually eliminate the initiation record but the act of forcing it onto stable storage. In PE, this is accomplished by having the coordinator of a transaction to write a non-forced initiation (or participants') record into the log buffer (in main memory) each time a new participant joins in the execution of the transaction. If the last initiation record that was written for the transaction is forced written (i.e., piggybacked) onto the stable log due to a subsequent forced log write, possibly on behalf of some other transaction, or a flush to the log buffer due to an overflow, PE behaves much like PrC for the rest of the protocol. Otherwise, PE behaves much like PrA.

Unlike PrC and PrA, PE uses a *flag* in the prepared message that a coordinator sends to each participant. The flag value is stored as part of the prepared record that is forced written at the participant's site. The purpose of the flag is to inform each participant about the used ACP and, consequently, to determine the behaviour of the participant, with the transaction, for the rest of the protocol. Indicating the used ACP in the flags is a necessity in PE not only for the determination of the behaviour of the participants for the rest of the protocol, during normal processing, but also for the correctness of recovery after a failure. That is, the flag value that is used with a transaction during normal commit processing determines the presumption that will be used by the coordinator in the event that the coordinator has already forgotten the transaction after a failure. Thus, presumption incompatibilities, at the transaction level, are resolved using the flag values that are stored as part of the prepared records at the participants.

Instead of having to use the same ACP by *all participants* for the commit processing of a transaction, DPr-2PC provides each participant with the flexibility to select its preferred protocol among PrA and PrC on a dynamic basis. That is, each participant has the freedom to select the most appropriate ACP variant with respect to performance, from its own point of view, before the beginning of the commit processing stage at the coordinator's site. In a manner similar to PE, this is accomplished using a flag, but the flag is used as part of the Ack of an operation that a participant executes on behalf of the transaction. This is in contrast to being part of the prepare-to-commit message that a coordinator sends to each participant, which is the case in PE. Of course, the chosen ACP by a participant can be overridden by the coordinator with a different flag value which is sent to the participant as part of the prepare-to-commit message. The details of this latter option are not exploited any further beyond mentioning it in DPr-2PC (Yu and Pu, 2007).

Once an ACP variant is decided with a participant on a specific transaction, in DPr-2PC, it is stored as part of the prepared log record of the transaction at the participant's site. After that, the protocol behaves much like PrAny except for recovering after a failure. In PrAny, the coordinator uses a single presumption about a forgotten transaction when responding to an inquiry message from a participant. This presumption always matches the presumption adopted by the protocol of the inquiring participant which is stored at the coordinator's site. This is in contrast to DPr-2PC in which a participant indicates the used protocol variant with a transaction in any inquiry message that it sends to the coordinator. This allows the coordinator to use the presumption of the protocol indicated in the inquiry message, which might change from one transaction to another, rather than using a single presumption with all transactions executed at the same participant. In both protocols, the used presumption by a coordinator always matches the actual final outcome of a forgotten terminated transaction.

The 1-2PC is another dynamic presumption protocol. It was designed as an attempt to achieve the best of the two worlds: The performance of 1PC protocols and the wide applicability of 2PC protocols. Specifically, 1-2PC is a combination of a 1PC protocol, namely IYV, and a 2PC protocol, namely PrC. The 1-2PC protocol integrates these two protocols in a dynamic fashion, depending on the behaviour of transactions and system requirements, in spite of their incompatibilities.

The main goal of 1-2PC is to achieve the performance of 1PC in the absence of *deferred consistency constraints* and the applicability of 2PC in the presence of such constraints. Unlike *immediate consistency constraints* that are validated as part of each individual database operation and their validation results are reflected on the results of each individual operation, deferred consistency constraints are validated at commit time of each transaction and their results are reflected on the execution of the transaction as a whole. Thus, the validation process of deferred consistency constraints at a participant needs to be synchronised at commit time and is triggered once the participant receives a prepare-to-commit message from the coordinator. Thus, the applicability of 1PC protocols is curtailed in systems that wish to utilise the option of deferred consistency constraints. This is because 1PC protocols eliminate the *explicit* voting phase of 2PC which serves as the triggering mechanism for the evaluation of deferred consistency constraints. 1-2PC was specifically designed to alleviate this shortcoming of 1PC protocols by allowing a 1PC participant in a transaction's execution to dynamically switch to 2PC once it recognises that the transaction is associated with deferred consistency constraints. Thus, 1-2PC supports the systems that wish to utilise the option of deferred consistency constraints, which is currently part of the structured query language (SQL) standards, and, at the same time, provides better commit processing performance over 2PC protocol variants.

In 1-2PC, each transaction starts as 1PC at each participant. The transaction continues this way until it executes the first operation that is associated with a deferred consistency constraint at the participant. Once this occurs, it means that the constraint needs to be validated at commit time of the transaction. For this reason, the participant switches to 2PC and sends an *unsolicited deferred consistency constraint* (UDCC) message to the coordinator. The UDCC is a flag that is set as part of the Ack for the successful execution of the operation that is associated with the deferred consistency constraint. When the coordinator receives such an Ack, it marks the participant as a 2PC participant in its protocol table.

At the end of a transaction, the coordinator knows which participants are 1PC and which participants have switched to 2PC. If all participants are 1PC (i.e., no participant has executed an operation that is associated with deferred constraints), the coordinator behaves as an IYV coordinator. On the other hand, if all participants are 2PC, the coordinator behaves as a PrC coordinator. When the participants are mixed 1PC and 2PC in a transaction's execution, the coordinator resolves the incompatibilities between the two protocols as follows:

1 It ‘talks’ IYV with 1PC participants and PrC with 2PC participants.

2 It initiates the voting phase with 2PC participants before making the final decision and propagating the final decision to all participants. (This is because a ‘no’ vote from a 2PC participant is a *veto* that aborts a transaction.)

3 It synchronises the timing at which it forgets the outcome of terminated transactions such that it forgets the outcome of a committed transaction when all 1PC participants acknowledge the commit decision, and the outcome of an aborted transaction when all 2PC participants acknowledge the abort decision.

AP^3 is also a dynamic presumption protocol. It extends 1-2PC by incorporating PrA besides PrC. That is, it integrates IYV, PrC and PrA. As in 1-2PC, AP^3 starts as 1PC at each participant and, only when necessary, it dynamically switches to 2PC. Thus, it achieves the performance advantages of 1PC whenever possible and, at the same time, the wide applicability of 2PC. However, when 2PC is to be used, unlike 1-2PC which has the ability to switch to only presumed commit, AP^3 has the ability to switch to the most appropriate 2PC variant to further enhance the performance of commit processing. In AP^3, the choice of 2PC depends on the anticipated results of the evaluation of the deferred consistency constraints. That is, if consistency constraints tend to be violated and consequently transactions tend to abort, presumed abort is used with 2PC participants. Otherwise, presumed commit is used. In this way, AP^3 supports deferred constraints without penalising those transactions that do not require them. Furthermore, as in 1-2PC, AP^3 achieves this advantage on a per participant basis within the same transaction in spite of the incompatibilities between 1PC and 2PC protocols.

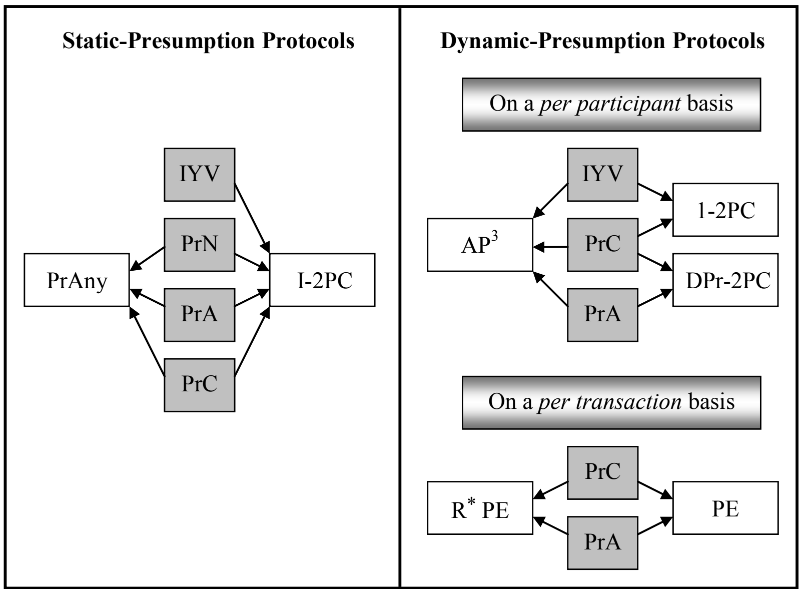
5.2.3 Comparison of integrated ACPs

Figure 6 summarises the different integrated ACPs and their basic component protocols. It also classifies each integrated protocol based on whether it is of a static nature, such as PrAny, or dynamic nature, such as PE.

The main difference between DPr-2PC and PE lies in the ability, of the former protocol, to switch from one protocol variant to another on a per participant basis rather than switching on a per transaction basis, which is the case in latter protocol. Thus, presumption incompatibilities arise within the same transaction in DPr-2PC whereas presumption incompatibilities arise across transactions in PE. Switching on a per participant basis makes DPr-2PC more suitable than PE for web service providers since it allows each service provider to retain some degree of autonomy with respect to its choices among ACPs.

On the other hand, the main difference between DPr-2PC and PrAny lies in how the coordinator obtains knowledge about the used protocol with a participant in a specific transaction and, consequently, the presumption that it uses in the event that it does not remember the transaction. In DPr-2PC, knowledge about the used protocol is obtained through the flag value that is stored as part of the transaction’s prepared record at the participant’s site which is included in any inquiry message that the participant sends to the coordinator. This value is not fixed and could change from one transaction to another. In contrast, knowledge about the used protocol by a participant, in PrAny, is fixed for all transactions and is stored at the coordinator’s site. Thus, the same presumption holds for *all* forgotten transactions that the *same* participant inquires about them.

Figure 6 Classifications of integrated ACPs


Static-Presumption Protocols
Dynamic-Presumption Protocols
On a *per participant* basis
On a *per transaction* basis
IYV
PrN
PrA
PrC
PrAny
I-2PC
AP3
1-2PC
DPr-2PC
R* PE
PE

Notes: Component protocols: PrN: basic (presumed nothing) two-phase commit [1976], PrA: presumed abort [1983], PrC: presumed commit [1983], IYV: implicit yes-vote [1995].

Integrated protocols: R* PE: system's R* presumed either [1983], PrAny: presumed any [1996], PE: presumed either [2002], 1-2PC: one-two phase commit [2004], DPr-2PC: dynamic presumption two-phase commit [2004], AP^3: adaptive participant's presumption protocol [2005], I-2PC: integrated two-phase commit [2008].

Another difference between DPr-2PC and PrAny lies in the storage of the used protocol. In DPr-2PC, there is no need for storing such knowledge forever. That is, once a transaction terminates at a participant, all its log records can be garbage collected when necessary. This is not the case in PrAny as knowledge about the used protocol by a participant has to be remembered, by the coordinator, so long as the participant remains part of the environment. This requirement is necessary, in PrAny, since there is no alternative way for the coordinator to know which protocol is used by each participant. This is in contrast to DPr-2PC where the flag mechanism reveals, to the coordinator, the used protocol by each participant.

Autonomy is yet another distinguishing factor between DPr-2PC and PrAny. Whereas DPr-2PC allows for more ACP choices for a participant to use, it forces each participant to use the same (integrated) protocol (i.e., DPr-2PC). This is not the case in PrAny where it integrates the already in-use protocols by the different participants without forcing any of them to use PrAny. However, it should be pointed out that DPr-2PC allows for backward compatibility in the sense that it allows for the existence of PrA participants without having them to use DPr-2PC in order to be integrated with DPr-2PC participants. This is accomplished by assuming that a participant in a transaction's execution is using PrA if it does not declare its used protocol with the transaction in an inquiry message. Backward compatibility is a further attempt to increase the degree of autonomy in DPr-2PC with respect to any pre-existing PrA participants.

6 Atomic commit protocol optimisations

Although any ACP variant can be considered as an optimisation to the basic 2PC where it is either optimised for the normal processing case (i.e., efficiency-geared) or for the failure case (i.e., reliability-geared), we distinguish between ACP *optimisation*s and ACP *variants* for two reasons. Firstly, an ACP optimisation cannot be used for atomic commitment independently without the use a hosting ACP whereas an ACP can be used independently without the use of any optimisation. Secondly, more than one optimisation can coexist in the same system without any possible conflict between them whereas no two ACPs can coexist in the same system without conflicts.

6.1 Most common optimisations

Several optimisations have been proposed to reduce the costs associated with ACPs (Samaras et al., 1995; Chrysanthis et al., 1998). Some of these optimisations are widely advocated in commercial database management systems and are currently parts of the specifications of distributed transaction processing standards. The most pronounced ACP optimisations include the *read-only, last agent, group commit, sharing the log, flattening the transaction tree* and *optimistic*.

The read-only optimisations can be considered as the most significant ones, given that read-only transactions are the majority in any general database system. The basic idea behind the read-only optimisations is that a read-only participant, a participant that has not performed any updates on behalf of a transaction, can be excluded from the decision phase of the transaction. This is because it does not matter whether the transaction is finally committed or aborted at a read-only participant to ensure the transaction's atomicity. This is, of course, so long as the transaction does not perform any further operations at the other participants after the read-only participant has terminated the transaction at its site (Lomet, 1993).

In the traditional read-only optimisation (Mohan et al., 1986), a read-only participant votes 'read-only' instead of a 'yes' and immediately releases all the resources held by the transaction without writing any log records. A 'read-only' vote allows a coordinator to recognise and discard the read-only participant from the rest of the protocol. In fact, the performance gains allowed by the traditional read-only optimisation provided the argument in favour of PrA, rather than PrC, to become the current choice of ACPs in the ISO OSI-TP (ISO, 1998) and X/Open DTP (X/Open Company Limited, 1996) distributed transaction processing standards, and commercial systems. This is because, given that read-only transactions are the majority in any general database system, it is cheaper to use PrA in combination with the traditional read-only optimisation than using PrC in combination with it as PrA does not require any forced log writes for read-only transactions whereas PrC still requires the forced initiation record at the coordinator's site. Thus, assuming that an optimisation can be used with two variants, it could be much more efficient to use the optimisation with one variant in comparison to using the same optimisation with the other, influencing the choice among ACP variants.

The *unsolicited update-vote* (UUV) (Al-Houmaily et al., 1997) is another read-only optimisation that further reduces the costs associated with read-only participants. Not only that, but it incurs the same costs when used with both PrA and PrC, supporting the arguments for PrC to be also included in future distributed transaction processing standards (Al-Houmaily et al., 1997). In UUV, each transaction starts as a read-only transaction at each participant and when the transaction performs the first update operation at a participant, the participant sends an 'unsolicited update-vote'. The 'unsolicited update-vote' is a flag that is set as part of the operation's Ack. Thus, at the end of a transaction, the coordinator knows all update participants and all read-only participants without having to explicitly pull the vote of each participant. In this way, the coordinator needs to inform each read-only participant that the transaction has terminated without requiring the participant to send back any reply message, reducing message complexity with read-only participants compared to the traditional read-only optimisation. For a completely read-only transaction, the costs of terminating the transaction in both PrA and PrC are the same, i.e., a single message to each participant without writing any log records neither at the coordinator's site nor at any of the participating sites. Thus, as transactions tend to commit when they reach their commit processing points and the costs associated with read-only transactions is the same in both PrA and PrC, it is cheaper to use PrC in combination with UUV than using PrA in combination with it. Again, this suggests that an optimisation could play an influential role in the choice among ACP variants.

The *last agent* optimisation (Samaras et al., 1995) has been implemented by a number of commercial systems to reduce the cost of commit processing in the presence of a *single remote* participant. In this optimisation, a coordinator first prepares itself and the nearby participants for commitment (fast first phase), and then delegates the responsibility of making the final decision to the remote participant. This eliminates the voting phase involving the remote participant. This same idea of delegating part of commitment (i.e., transferring the commit responsibilities) from one site to another has been also used to reduce blocking, for example, in *open commit protocols* (OCPs) (Rothermel and Pappe, 1993) and *IYV with a commit coordinator* (IYV-WCC) (Al-Houmaily and Chrysanthis, 1996b).

The *group commit* optimisation (DeWitt et al., 1984; Gawlick and Kinkade, 1985) has been also implemented by a number of commercial products to reduce log complexity. In the context of centralised database systems, a commit record pertaining to a transaction is not forced on an individual basis. Instead, a single force-write to the log is performed when a number of transactions are to be committed or when a timer has expired. In the context of distributed database systems, this technique is used at the participants' sites *only* for the commit records of transactions during commit processing. The *lazy commit* optimisation (Gray and Reuter, 1993) is a generalisation of the *group commit* in which not only the commit records at the participants are forced in a group fashion, but *all* log records are lazily forced written onto stable storage during commit processing. Thus, the cost of a single access to the stable log is amortised among several transactions. The s*haring of log* between the TM and RMs (Samaras et al., 1995) at a site is another optimisation that takes advantage of the sequential nature of the log to eliminate the need of force writing the log by the RMs.

The *flattening of the transaction tree* optimisation (Samaras et al., 1995) is targeted for the MLTE model and is a big performance winner in distributed transactions that contain deep trees. It can reduce both the message and log complexities of an ACP by transforming the transaction execution tree of *any* depth into a two-level commit tree at commit initiation time. In this way, the root coordinator sends coordination messages directly to, and receives messages directly from, any participant. This leads to avoiding propagation delays and sequential forcing of log records. The flattening of the transaction tree optimisation was the base for the design of the *restructured presumed commit* (ReSPrC) protocol (Al-Houmaily et al., 1997). ReSPrC identifies all the participants in a transaction's execution tree during the execution phase of the transaction and transforming the multi-level structure of the tree into a two-level commit tree at commit processing time. In this way, ResPrC reduces the number of initiation records that needs to be forced written at each cascaded coordinator, when compared to ML-PrC, to only a single record that is forced written at the root coordinator of the tree. *Restructuring-the-commit-tree around update-participants* (RCT-UP) (Samaras et al., 2003) is an enhancement to the flattening technique that flattens only update participants (i.e., participants that have executed update operations on behalf of the transaction), thereby, connecting them directly to the root coordinator while leaving read-only participants connected in a multi-level manner. This is to reduce the effects of the communication delays on the overall system performance in systems that do not support simultaneous message multicasting to all participants.

Another optimisation is *optimistic* (OPT) (Gupta et al., 1997) which can enhance the overall system performance by reducing blocking arising out of locks held by prepared transactions. OPT shares the same assumption as PrC, that is, transactions tend to commit when they reach their commit points. Under this assumption, OPT allows a transaction to *borrow* data that have been modified by another transaction that has entered a prepared to commit state and has not committed. Thus, the *strictness* property of the execution of transactions that is required for recovery is relaxed in OPT. Consequently, a borrowing transaction is aborted if the lending transaction is finally aborted. However, to limit the magnitude of cascaded aborts of borrowing transactions and, consequently, complex recovery, OPT allows only one-level borrowing of data. That is, a borrowing transaction is not allowed to lend its data to any other transaction. OPT can be used in conjunction with most ACPs including 2PC, PrA, PrC and 3PC, resulting in OPT-2PC, OPT-PrA, OPT-PrC and OPT-3PC, respectively (Gupta et al., 1997).

7 Summary and conclusions

Basic reliability guarantees that provide for the assurances of deterministic outcomes in the presence of faults, similar to the *atomic commit protocols* (ACPs) that facilitate the 'atomicity' property of transactions in distributed database systems, represent a necessary requirement for the development of advanced software application systems. For this reason, it is imperative to understand the basic problem of *atomic commitment* in distributed database systems and to know its current dimensions as well as its proposed solutions. This represented the focus of this paper through a comprehensive treatment to the problem along the following lines:

1 thoroughly explaining the well-known *two-phase commit* (2PC) protocol and its most commonly known two variants, namely *presumed commit* and *presumed abort*

2 classifying ACPs into two categories: *efficiency-geared* and *reliability-geared*, and discussing the most significant ones in each category

3 discussing the sources of incompatibilities among ACPs and the proposed solutions that overcome these incompatibilities as well as classifying the proposed solutions

4 distinguishing between ACP variants and ACP optimisations, and discussing the most widely known ACP optimisations.

Tracing the problem of atomic commitment in distributed database systems provides further stimulation to this ever-evolving area of research and establishes a foundational stage for investigating and developing more elaborate and novel solutions to the problem. Some of these solutions are likely to satisfy the reliability needs of future generations' application systems in a cost-effective manner. Besides that, broadening the applicability scope of some of the presented solutions to other 'non-classical' distributed database environments represents a major topic in this area of research.

References

Al-Houmaily, Y. (2005) 'On interoperating incompatible atomic commit protocols in distributed databases', *Proc. of the 1st IEEE Int'l Conf. on Computers, Communications, and Signal Processing*, Kuala Lumpur, Malaysia.

Al-Houmaily, Y. (2008) 'Incompatibility dimensions and integration of atomic commit protocols', *Int'l Arab J. of Inf. Tech.*, Vol. 5, No. 4.

Al-Houmaily, Y. and Chrysanthis, P. (1996a) 'Dealing with incompatible presumptions of commit protocols in multidatabase systems', *Proc. of the ACM SAC*, Philadelphia, USA.

Al-Houmaily, Y. and Chrysanthis, P. (1996b) 'The implicit yes-vote commit protocol with delegation of commitment', *Proc. of the 9th ISCA Int'l Conf. on Parallel and Distributed Computing Systems*, Dijon, France.

Al-Houmaily, Y. and Chrysanthis, P. (1999) 'Atomicity with incompatible presumptions', *Proc. of the 18th ACM PODS.*

Al-Houmaily, Y. and Chrysanthis, P. (2000) 'An atomic commit protocol for gigabit-networked distributed database systems', *J. of Systems Architecture*, Vol. 46, pp.809–833.

Al-Houmaily, Y. and Chrysanthis, P. (2004a) '1-2PC: the one-two phase atomic commit protocol', *Proc. of the ACM SAC*, Nicosia, Cyprus.

Al-Houmaily, Y. and Chrysanthis, P. (2004b) 'ML-1-2PC: an adaptive multi-level atomic commit protocol', *Proc. of the 8th East European Conf. on the Advances in Databases and Inf. Systems*, Budapest, Hungary.

Al-Houmaily, Y., Chrysanthis, P. and Levitan, S. (1997) 'An argument in favor of the presumed commit protocol', *Proc. of the 13th ICDE.*

Attaluri, G. and Salem, K. (2002) 'The presumed-either two-phase commit protocol', *IEEE TKDE*, Vol. 14, No. 5, pp.1190–1196.

Awan, I. and Younas, M. (2004) 'Analytical modeling of priority protocol for reliable web applications', *Proc. of the ACM SAC*, Nicosia, Cyprus.

Chrysanthis, P., Samaras, G. and Al-Houmaily, Y. (1998) 'Recovery and performance of atomic commit processing in distributed database systems', in Kumar, V. and Hsu, M. (Eds.): *Recovery Mechanisms in Database Systems*, pp.370–416, Prentice Hall.

DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M. and Wood, D. (1984) 'Implementation techniques for main memory database systems', *Proc. of the ACM SIGMOD*, pp.1–8.

Gawlick, D. and Kinkade, D. (1985) 'Varieties of concurrency control in IMS/VS fast path', *IEEE Database Engineering*, Vol. 8, No. 2.

Gray, J. (1978) 'Notes on database operating systems', in Bayer, R., Graham, R.M. and Seegmuller, G. (Eds.): *Operating Systems: An Advanced Course, LNCS*, Vol. 60, pp.393–481, Springer-Verlag.

Gray, J. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Inc., USA.

Gupta, R., Haritsa, J. and Ramamritham, K. (1997) 'Revisiting commit processing in distributed database systems', *Proc. of the ACM SIGMOD*.

Hammer, M. and Shipman, D. (1980) 'Reliability mechanisms for SDD-1: a system for distributed databases', *ACM TODS*, Vol. 8, No. 4, pp.431–466.

Haritsa, J., Ramamritham, K. and Gupta, R. (2000) 'The PROMPT real-time commit protocol', *IEEE Trans. Parallel and Dist. Sys.*, Vol. 11, No. 2.

ISO (1998) 'Open systems interconnection – distributed transaction processing – Part 1: OSI TP model', ISO/IEC 10026-1.

Lamport, L., Shostak, R. and Pease, M. (1982) 'The Byzantine generals problem', *ACM Trans. Prog. Lang. and Sys.*, Vol. 4, No. 3, pp.382–401.

Lampson, B. (1981) 'Atomic transactions', in Lampson, B., Paul, M. and Siegert, H.J. (Eds.): *Distributed Systems: Architecture and Implementation – An Advanced Course, LNCS*, Vol. 105, pp.246–265, Springer-Verlag.

Lampson, B. and Lomet, D. (1993) 'A new presumed commit optimization for two phase commit', *Proc. of the 19th Int'l Conf. on VLDB*, pp.630–640.

Lee, I. and Yeom, H. (2002) 'A single phase distributed commit protocol for main memory database systems', *Proc. of the 6th Int'l Parallel and Distributed Processing Symposium*.

LeLann, G. (1981) 'Error recovery', in Lampson, B., Paul, M. and Siegert, H.J. (Eds.): *Distributed Systems: Architecture and Implementations – An Advanced Course, LNCS*, Vol. 105, pp.371–376, Springer-Verlag.

Lomet, D. (1993) 'Using timestamping to optimize two phase commit', *Proc. of the 2nd Int'l Conf. on Parallel and Distributed Systems*, pp.48–55.

Microsoft Corporation (2007) *Microsoft .NET Framework 3.5 Administrator Development Guide*, available at http://msdn.microsoft.com/en-us/library/cc160717.aspx.

Mohan, C., Britton, K., Citron, A. and Samaras, G. (1993) 'Generalized presumed abort: marrying presumed abort and SNA's LU 6.2 commit protocols', *Proc. of the 5th Int'l Workshop on High Performance Transaction Systems*.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. and Schwarz, P. (1992) 'ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging', *ACM TODS*, Vol. 17, No. 1, pp.94–162.

Mohan, C., Lindsay, B. and Obermarck, R. (1986) 'Transaction management in the R* distributed data base management system', *ACM TODS*, Vol. 11, No. 4, pp.378–396.

Nouali-Taboudjemat, N., Boukantar, L. and Drias, H. (2007) 'Performance evaluation of atomic commit protocols for mobile transactions', *Int'l J. of Intelligent Information and Database Systems*, Vol. 1, No. 2, pp.122–155.

Object Management Group, Inc. (2003) *Transaction Service Specification*, version 1.4, available at http://www.omg.org/docs/formal/03-09-02.pdf.

Organization for the Advancement of Structured Information Standards (OASIS) (2007) *Web Services Transaction (WS-Transaction)*, version 1.1, available at http://www.oasis-open.org/committees/ws-tx.

Pu, C., Leff, A. and Chen, S. (1991) 'Heterogeneous and autonomous transaction processing', *IEEE Computer*, Vol. 24, No. 12, pp.64–72.

Rocha, T., Arntsen, A-B., Eidsvik, A., Toledo, M. and Karlsen, R. (2007) 'Promoting levels of openness on component-based adaptable middleware', *Proc. 6th Int'l Workshop on Adaptive and Reflective Middleware*.

Rothermel, K. and Pappe, S. (1993) 'Open commit protocols tolerating commission failures', *ACM TODS*, Vol. 18, No. 2, pp.289–332.

Samaras, G. and Nikolopoulos, S. (1995) 'Algorithmic techniques incorporating heuristic decisions to commit protocols', *Proc. of the 21st Euromicro Conf.*, Como, Italy.

Samaras, G., Britton, K., Citron, A. and Mohan, C. (1995) 'Two-phase commit optimizations in a commercial distributed environment', *Distributed and Parallel Databases*, Vol. 3, No. 4, pp.325–361.

Samaras, G., Kyrou, G. and Chrysanthis, P. (2003) 'Two-phase commit processing with restructured commit tree', *Proc. of the Panhellenic Conf. on Informatics, LNCS*, Vol. 2563, pp.82–99.

Skeen, D. (1981) 'Non-blocking commit protocols', *Proc. of the ACM SIGMOD*, pp.133–142.

Skeen, D. and Stonebraker, M. (1983) 'A formal model of crash recovery in a distributed system', *IEEE Trans. Software Eng.*, Vol. 9, No. 3.

Spiro, P., Joshi, A. and Rengarajan, T. (1993) 'Designing and optimized transaction commit protocol', *Digital Technical Journal*, Vol. 3, No. 1.

Stamos, J. and Cristian, F. (1993) 'Coordinator log transaction execution protocol', *Distributed and Parallel Databases*, Vol. 1, No. 4, pp.383–408.

Stonebraker, M. (1979) 'Concurrency control and consistency of multiple copies of data in distributed INGRES', *IEEE Trans. Software Eng.*, Vol. 5, No. 3.

Sun Microsystems (2006) Java™ Platform, Enterprise Edition (Java EE) Specification, v5.

Tal, A. and Alonso, R. (1994) 'Commit protocols for externalized-commit heterogeneous databases', *Distributed and Parallel Databases*, Vol. 2, No. 2.

X/Open Company Limited (1996) Distributed Transaction Processing: Reference Model, Version 3 (X/Open Doc. No. 504).

Yu, W. and Pu, C. (2007) 'A dynamic two-phase commit protocol for adaptive composite services', *Int'l J. of Web Services Research*, Vol. 4, No. 1.

Notes

1 Notice that the other sites might lag in their states because of systems' delays such as queuing and network delays.