Schemes for Atomicity in Heterogeneous Database Environments

Yousef J. Al-Houmaily

Department of Computer and Information Programs
Institute of Public Administration, Riyadh, Saudi Arabia

Keywords *Internet Transactions, Atomic Commit Protocols, Database Integration, Distributed Database Systems, Systems' Reliability.*

Abstract *Advanced software application systems such as electronic commerce and electronic services, web-based applications and multi-organizational workflows contain database transactions that tend to traverse multiple heterogeneous database sites. Such transactions constitute basic building blocks for these advanced applications because of their attractive consistency and reliability characteristics. Thus, it is imperative to provide universal transactional support for these applications and, in particular, guaranteeing the atomicity property of distributed transactions in spite of the heterogeneity of the database sites that they access. Whereas the atomicity property is well understood and relatively simple to realize in (homogeneous) distributed database systems, it is practically much harder to realize in heterogeneous environments such as the Internet and multidatabase systems. This is because of the heterogeneity characteristics of the constituent database sites, on one hand, and their autonomy requirements, on the other.*

Based on the above, it is very important to investigate the different schemes that have been proposed to characterize atomicity in heterogeneous environments and classify them in a form of taxonomy, which is the focus of this paper. Such taxonomy is a necessary first step towards a better understanding to atomicity in heterogeneous environments and a framework that can be used for the development of new and more flexible recovery methods for highly reliable Internet applications. The taxonomy proposed in this paper is essentially based on the assumptions that the different schemes make about the mechanisms used for atomicity by the constituent database sites.

Introduction

Database transactions exhibit attractive consistency and reliability guarantees that make them very appealing basic building blocks for the development of advanced software application systems. These applications include electronic commerce and electronic services, web-based applications and multi-organizational workflows (to name just a few). Thus, it is imperative to support *universal transactional access* and, in particular, guaranteeing the *atomicity* property of transactions across heterogeneous database sites.

The atomicity property ensures that each transaction is executed as a single, indivisible unit of work that is either performed to its completion or not at all. That is, each transaction either *commits* (i.e., executes successfully to its completion) or *aborts* (i.e., fails at some point during its execution). When a transaction commits, all its effects become permanent on the state of the database whereas, when the transaction aborts, all its effects are obliterated from the state of the database as if it had never existed.

In a centralized database system, the atomicity property is commonly satisfied by the recovery protocol that is implemented as part of its database management system. On the other hand, in a *distributed database system* (DDBS), this property is commonly guaranteed by complementing the recovery protocol by a form of an *atomic commit protocol* (ACP). The *two-phase commit* (2PC) protocol (Gray, 1978; Lampson, 1981) is the simplest and most widely implemented ACP .

Although 2PC ensures atomicity of distributed transactions and allows for the independent recovery of database sites, it consumes a substantial amount of execution time during normal transaction processing that adversely affects the performance of the system. This is due to the costs associated with *message complexity* (i.e., the number of messages used for coordinating the actions of the different database sites) and *log complexity* (i.e., the frequency at which information is stored onto the stable storage of the participating sites). For this reason, there has been a continuous interest in developing more efficient ACPs and optimizations, e.g., (Gupta, et. al., 1997; Abdallah et. al., 2002; Attalui and Salem, 2002; Samaras et. al., 2003; Al-Houmaily and Chrysanthis, 2004 (a); Al-Houmaily and Chrysanthis, 2004 (b); Al-Houmaily, 2005; Yu and Pu, 2007). Most notable results that aim at reducing the cost of commit processing are *one-phase commit* (1PC) protocols, such as *implicit yes-vote* (IYV) (Al-Houmaily and Chrysanthis, 2000) and *coordinator log* (CL) (Stamos and Cristian, 1993).

Whereas the atomicity property is well understood and relatively simple to realize in (homogeneous) distributed database systems, it is practically much harder to realize in heterogeneous environments such as the Internet and *multidatabase systems* (MDBSs). This is because each of the database sites is pre-existing and, most probably, heterogeneous and autonomous. That is, each database site is pre-existing in the sense that it could be operational, supporting its own local users and applications prior to joining the system. A database site could be also heterogeneous in the sense that it might be using, for example, a different data model or transaction management mechanisms from the other sites in the environment. Furthermore, a database site could be autonomous in the sense that it might not be willing to modify the way in which it operates or relinquish any control information over its database to any other organization. In spite of these difficulties, there is a greater need than ever before to interoperate the different database sites in a practical manner especially in the context of the rapidly growing number of advanced database applications that characterize today's Internet. A key requirement of these application systems is providing recovery guarantees in the presence of failures (Barga, 2004).

The focus of this paper is on the atomicity property of transactions in heterogeneous environments. More specifically, it *investigates* the different approaches that have been proposed to characterize atomicity in MDBSs, as a representative to heterogeneous database environments, and *classifies* them in a form of taxonomy. The proposed taxonomy is essentially based on the assumptions that the different approaches make about the mechanisms used by the constituent database sites to ensure atomicity. Thus, it can be viewed as a first step towards a better understanding to the relationships among the different schemes that ensure atomicity in heterogeneous environments and a framework for developing new and more flexible recovery methods for advanced distributed software applications. It should be noted that, given the continuous advancements in this area of research, the taxonomy proposed in this work incorporates and extends our previous results that appeared in (Al-Houmaily, 1997).

The rest of the paper is structured as follows: Section “Background” contains some background material from the database literature and a three-dimension classification to distributed database systems, emphasizing the MDBS environment as a representative to heterogeneous environments. It also discusses the need for ACPs in distributed database systems. Section “Atomic Commit Protocols and Dimensions of Correctness” describes the *basic* 2PC protocol and draws a distinction between *functional correctness* and *operational correctness* of ACPs. It also describes the most commonly known 2PC variants, namely *presumed abort* and *presumed commit*. Section “Taxonomy of Atomic Commitment in MDBSs” contains the details of our taxonomy to the schemes that ensure atomicity in heterogeneous database environments while the last section summarizes and concludes the paper.

Background

In a traditional database system, a *database management system* (DBMS) is employed to provide uniform access to the data objects stored in the database and to ensure their consistency. Application programs and users interact with the data by submitting *transactions* to the DBMS which executes them against the database.

In the traditional transaction model, transactions are *atomic*. That is, each transaction is executed as a single logical unit of work where all its effects are either reflected on the state of the database or not at all. In this model, a transaction is a partial order set of *events* that starts by invoking a *Begin* significant event to indicate the start of the transaction. The “Begin” event is followed by a sequence of operation events. For example, an operation event could be a *Read* or a *Write* operation on a data item. When a transaction invokes a “Read” operation on an object, it retrieves the state (i.e., the value) of the data object whereas, when the transaction invokes a “Write” operation on an object, it updates the state of the object. Thus, *significant events* are used for the management of the database while *operation events* are used for the actual manipulation of the data stored in the database.

When a transaction finishes its execution, it invokes a significant event that is either a *Commit* or an *Abort* to indicate its intention to install the changes that it has made on the state of the database. If the event is a “Commit”, it means that the transaction wants to install all its effects on the state of the database; whereas, if the event is an “Abort”, it means that the transaction wants to cancel (i.e., rollback) it effects from the state of the database.

To maximize transaction throughput and resource utilization in database systems, transactions are executed concurrently, allowing them to interleave their operations on the database. Since the concurrent execution of transactions may cause them to interfere with each other over the data objects, data consistency might be violated. Data consistency might be also violated in the case of failures. This is because transactions might end up partially executed, producing unpredictable results. As an example, consider a fund transfer transaction that fails after it debits one account but before it credits the other one. In this example, the total credit in the two accounts becomes invalid after the transaction failure. Failures could be due to software, such as an operating system failure, or hardware, such as a power outage.

In traditional centralized database systems, data inconsistencies that are due to failures and concurrency are prevented by satisfying the, commonly known, ACID properties (Gray, 1978; Gray, 1981; Haerder and Reuter, 1983) which are associated with transactions. The ACID properties are: (1) *Atomicity*, (2) *Consistency*, (3) *Isolation* and (4) *Durability*. Atomicity ensures that, regardless of failures, all the operations of a transaction are treated as a single, indivisible, atomic unit of work that is either performed when the transaction *commits* (i.e., finishes its execution successfully) or not at all when the transaction *aborts* (i.e., fails). Consistency is a requirement that is placed on transactions in the sense that each transaction is a computation that maintains the database consistency constraints. Isolation allows transactions to execute concurrently in the system without violating the consistency of the database. Durability ensures that the effects of committed transactions are made permanent on the state of the database, surviving any subsequent failures.

A database management system usually ensures the ACID properties of transactions by combining two algorithms. The first one satisfies the isolation property and is referred to as *concurrency control protocol*, whereas the second algorithm satisfies the atomicity and durability properties and is referred to as *recovery protocol*. The consistency property is, commonly, ensured by designing transactions such that each one of them preserves the consistency of the database within its boundaries (i.e., its begging and its end).

To set the stage for the rest of the paper, the next section describes the system model of distributed database systems that will be used in the paper, emphasizing the multidatabase system model as a representative heterogeneous database environment. The system model that is used in this paper is similar to the one presented in (Al-Houmaily, 1997).

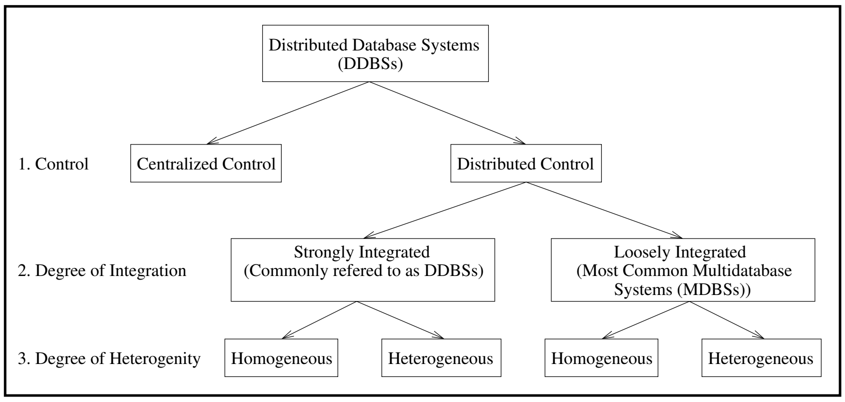

Distributed Database Systems (DDBSs)
1. Control
Centralized Control
Distributed Control
2. Degree of Integration
Strongly Integrated (Commonly refered to as DDBSs)
Loosely Integrated (Most Common Multidatabase Systems (MDBSs))
3. Degree of Heterogenity
Homogeneous
Heterogeneous
Homogeneous
Heterogeneous

Figure 1: Classification dimensions of distributed database systems.

System Model

A *distributed database system* (DDBS) is a collection of database sites that are interconnected via a communication network. The data objects in a DDBS might be stored as disjoint partitions at different sites, replicated across sites or a combination of both. Application programs and users interact with the distributed data by means of *distributed transactions*.

Individual database sites are responsible for the management of their databases while the control and coordination of transaction processing that ensure global data consistency can be either *centralized* or *distributed.*

As shown in Figure 1, the different DDBSs can be classified based on three dimensions. These dimensions are: (1) *locality of control*, (2) *degree of integration*, and (3) *degree of heterogeneity* (Al-Houmaily, 1997). In a centralized control DDBS, only a particular site is responsible for the consistency of the distributed data by controlling and coordinating the transaction processing activities across the different sites. On the other hand, in a distributed control DDBS, the different sites share the responsibility of control over the execution of transactions and interact cooperatively to achieve data consistency.

The second dimension, i.e., degree of integration, refers to degree of information sharing. A distributed control DDBS can be regarded as either *strongly integrated* or *loosely integrated*. In a strongly integrated, distributed control DDBS, each database site shares sufficient control information regarding the state of its database and the state of the locally executing transactions with the other sites to ensure global data consistency. On the other hand, in a loosely integrated, distributed control DDBS, each site preserves some degree of *autonomy* and it might not be willing to exchange any control information with any other site. A special type of loosely integrated, distributed control DDBSs is *multidatabase systems* (MDBSs). An MDBS responds to the needs of different human organizations that wish to interoperate their database systems that are already in service supporting their own local applications and users, without making any modifications to the way they operate, a scenario that is similar to the database sites that exist on the Internet today.

The third dimension, i.e., degree of heterogeneity, refers to the similarity of the mechanisms used by the database management systems of the different sites. That is, the database sites in a distributed control DDBS, whether strongly or loosely integrated, can be *homogeneous* or *heterogeneous*. In the former case, the database sites employ identical mechanisms for the management of data and transaction processing whereas, in the latter case, the different sites employ different mechanisms for either the management of data, transaction processing or both.

A Representative Heterogeneous Database Environment: Multidatabase Systems

As mentioned above, an MDBS is a special type of distributed database systems that interoperates multiple, pre-existing and, most probably, heterogeneous and autonomous database sites. Each database site is pre-existing in the sense that it is operational prior to joining the multidatabase system, supporting its own local applications and users. A database site may be also heterogeneous in the sense that it might be employing mechanisms that differ from those employed by the other database sites for the management of its data. Finally, each database site is characterized by its autonomy in the sense that it might not be willing to share any control information regarding the way in which it manages its data with the outside world.

Ideally, an MDBS allows each database system to continue to operate in an independent fashion without any changes to existing databases, applications or database management systems. Thus, an MDBS enables for data sharing through the development of advanced database applications that access multiple database sites and, at the same time, preserves the autonomy of the constituent database sites.

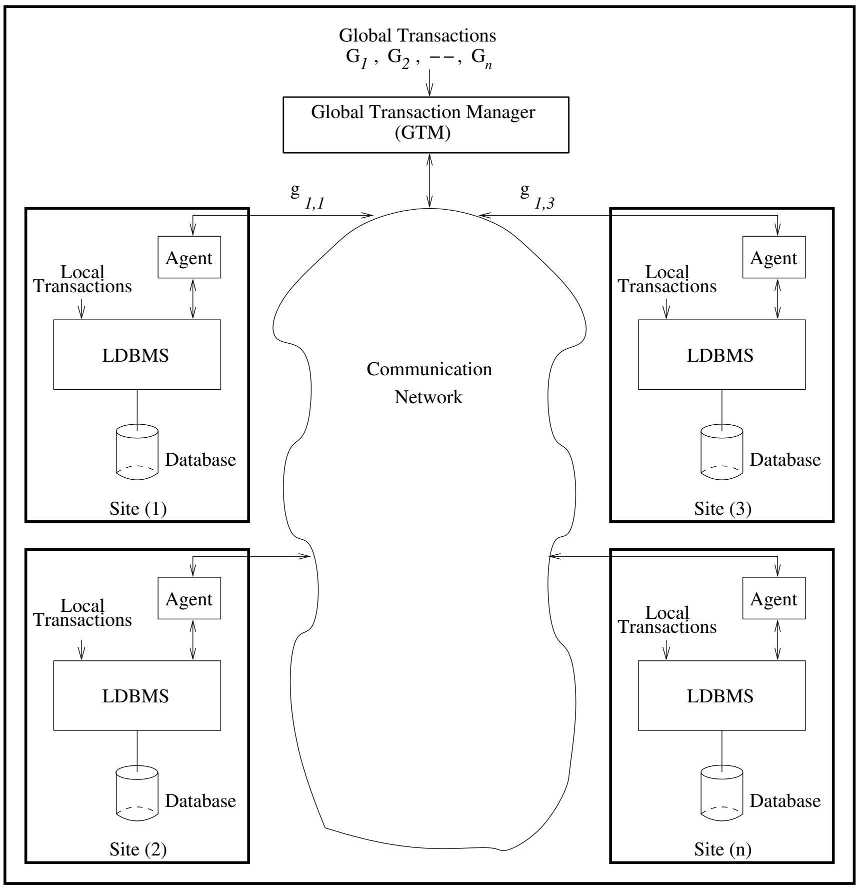


Figure 2: The multidatabase system model.

Figure 2 shows the two software components that are commonly modeled in MDBSs. These two components are: (1) a *global transaction manager* (GTM) and (2) a set of *agents*. The GTM is responsible for the management of global transactions that facilitate interoperability across the different database sites; whereas each agent is responsible for the different aspects of the execution of global transactions at its site.

Two types of transactions are also shown in Figure 2. These two types are (Al-Houmaily, 1997):

1. *Local transactions* that access data located at only a single database. This type of transactions corresponds to the transactions submitted by local users and applications and executes under the control of the *local database management systems* (LDBMSs) without any intervention from the MDBS.

2. *Global transactions* that access data located at multiple database sites under the control of the MDBS.

Distributed Transactions and Atomicity

In a DDBS, irrespective of its classification, each transaction is associated with a *coordinator*. The coordinator is responsible about coordinating the different aspects of the transaction execution and is assumed, without loss of generality, to be the *transaction manager*, a software component that is part of the database management system (DBMS), at the site where the transaction is initiated. In the context of MDBSs, the coordinator of global transactions is the GTM.

As in the case of (homogeneous) distributed database systems, a global transaction is decomposed by the GTM into several *subtransactions*, each of which executes as a local transaction at some site. The data distribution is transparent to submitted global transactions. That is, a transaction accesses data by submitting its data operations to the GTM without knowing about the locations of the data objects in the environment. It is the responsibility of the GTM to determine the appropriate participant's site to which it submits each data operation it receives from the transaction. Hence, as shown in Figure 2, a global transaction G_i is decomposed by the GTM into several subtransactions, $g_{i,j}$, each of which executes at a single database site where "*i*" represents the identity of the global transaction and "*j*" represents the site number, respectively. For example, the subtransactions of G_1, in Figure 2, are $g_{1,1}$ and $g_{1,3}$.

Unlike centralized database systems, atomicity of transactions has to be synchronized across all participating sites in a distributed database environment. To illustrate the need for such synchronization, assume that G_1 is the fund transfer transaction that was mentioned earlier in Section "Background". Furthermore, assume that $g_{1,1}$ is the debit part of the transaction while $g_{1,3}$ is the deposit part. Now, consider the case where the debit part of the fund transfer transaction (i.e., $g_{1,1}$) has finished its execution and the deposit part (i.e., $g_{1,3}$) started and finished its execution after that. Then, the fund transfer transaction submitted its commit primitive to the GTM which, in turn, forwarded commit messages to both participants (i.e., $Site_1$ and $Site_3$). At this point, assume that $Site_1$ has received the commit message and committed the transaction at its site while $Site_3$ failed just before receiving the commit message. In this case, $Site_3$ will not find a commit record for the transaction in its log during its recovery procedure and will consider the transaction as aborted, obliterating all its effects. Thus, in this example, the atomicity of the transaction has been violated because the transaction has ended up committing at $Site_1$ and aborting at $Site_3$, violating data consistency.

To prevent such atomicity violations of global transactions in the presence of possible site and communication failures, an *atomic commit protocol* (ACP) is usually implemented as part of any DDBS. This is in order to coordinate the commitment of distributed transactions across multiple database sites. The *two-phase commit* protocol (2PC) (Gray, 1978; Lampson, 1981) is the simplest and most widely used ACP .

Atomic Commit Protocols and Dimensions of Correctness

In this section, we discuss the 2PC protocol and its most commonly known variants, namely *presumed abort* (PrA) and *presumed commit* (PrC) (see Chrysanthis et. al., 1998 for a survey of the most commonly known 2PC variants). During the discussion of the basic 2PC, we define two dimensions for correctness of ACPs. The first one is *functional correctness* while the second one is *operational correctness*. These two dimensions of correctness will help in understanding the differences between the proposals that ensure atomicity in MDBSs when we present our taxonomy.

The Two-Phase Commit Protocol

As shown in Figure 3, the 2PC protocol, which is initiated by a coordinator after a transaction has finished its execution, consists of two phases. These two phases are the *voting phase* and the *decision phase*. During the voting phase, the coordinator requests all participating sites to *prepare to commit* whereas, during the decision phase, the coordinator either commits the transaction if *all* participants are prepared-to-commit (voted "yes"), or aborts the transaction if any participant has decided to abort (voted "no"). A participant votes "yes" only if can comply with a commit final decision. Otherwise, it aborts the transaction and sends a "no" vote. When a participant votes "yes", it enters a prepared-to-commit state whereby it can neither commit nor abort the transaction until it receives the coordinator's final decision. When a participant receives the final decision, it complies with the decision

.

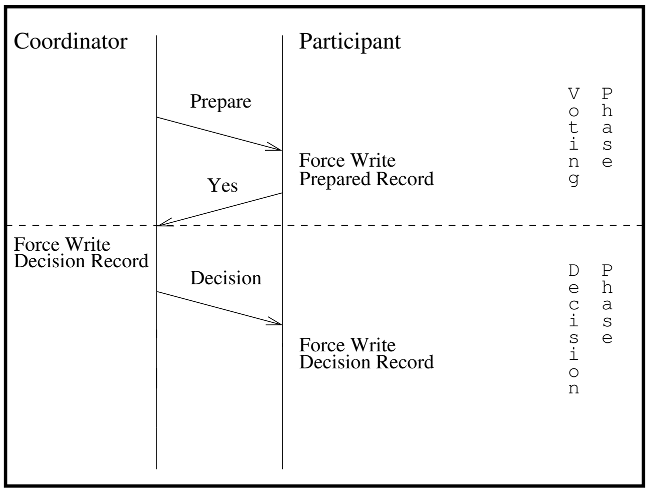

Coordinator
Participant
Prepare
Force Write Prepared Record
Yes
Voting Phase
Force Write Decision Record
Decision
Force Write Decision Record
Decision Phase

Figure 3: A functionally correct two-phase commit protocol.

Since the objective of 2PC is to achieve atomicity of transactions in the presence of possible system and communication failures, the protocol requires that the coordinator and each participant to record sufficient information about the progress of the protocol in their logs. Specifically, the coordinator is required to *force write* a decision record prior to sending out the final decision. Similarly, each participant is required to *force write* a prepared record before sending its “yes” vote, and a decision record after receiving the final decision from the coordinator. Since a forced write of a log record ensures that the record is written onto a stable storage that survives system failures, the coordinator and each participating site can remember the progress of 2PC protocol in the event of a failure.

The above form of 2PC satisfies *functional correctness*. That is, it ensures that all participating sites reach the same final decision about a transaction and enforce the same decision even in the presence of site or communication failures. This is because a coordinator, after making a final decision for a transaction, will always remember the outcome of the transaction and will be able to respond to any inquiry message from a prepared-to-commit participant with the same final outcome for the transaction. Similarly, each prepared-to-commit participant will always remember the transaction until it receives the final decision from the coordinator.

The functional correctness of ACPs can be stated formally as follows:

Definition 1: An ACP is *functionally correct* if and only if all participating sites reach consistent decisions regarding the outcome of transactions and regardless of failures.

However, achieving *only* functional correctness by an ACP is not sufficient for the practicality of the protocol. In the above form of 2PC, once a coordinator of a transaction has made a final decision, it has to remember the outcome of the transaction *forever*. This is because, the coordinator, after making the final decision for the transaction, has no knowledge about whether all participants have received the decision *safely* due to the possible failures. Remembering the outcome of transactions forever has the consequence of not being able to garbage collect the log records of terminated transactions, curtailing the system’s operation due to the unlimited growth of the coordinator’s log. Consequently, such a protocol curtails the system’s operation on the long run. This leads us to define another criterion that captures the practicality of any ACP besides functional correctness. We call such a criterion *operational correctness* [1].

Definition 2: An ACP is *operationally correct* if and only if

1. All participating sites reach consistent decisions regarding the outcome of transactions and regardless of failures (i.e., the protocol is functionally correct).

2. All participating sites can, eventually, discard all information pertaining to terminated transactions from their protocol tables and garbage collects their logs.

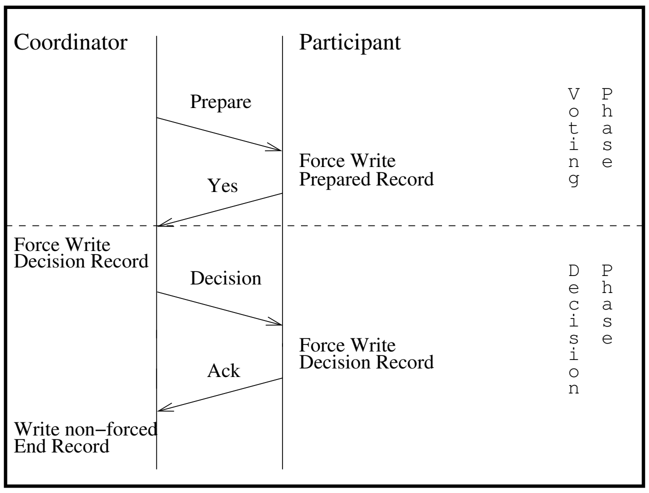

Coordinator
Participant
Prepare
Force Write Prepared Record
Yes
Voting Phase
Force Write Decision Record
Decision
Force Write Decision Record
Ack
Decision Phase
Write non−forced End Record

Figure 4: An operationally correct two-phase commit protocol.

As shown in Figure 4, the original 2PC protocol (Gray, 1978; Lampson, 1981) captures our operational correctness criterion by forcing each participant to acknowledge the coordinator's final decision after receiving and enforcing the decision locally at its site. Once the coordinator has received acknowledgments from all participating sites, the coordinator writes an *end* log record in its log buffer in main memory without forcing the log record onto the stable log and, then, forgets the transaction (see Al-Houmaily et. al., 1997 for a detailed description of the 2PC protocol). Thus, the final round of (acknowledgment) messages and the end log record are used for bookkeeping purposes that are not necessary from functional correctness point of view, but certainly a necessity from operational correctness point of view.

Two-Phase Commit Variants

The basic 2PC is also referred to as the *presumed nothing* 2PC protocol (PrN) (Lampson and Lomet, 1993) 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, in the case of a coordinator's failure, there is a hidden presumption in PrN by which the coordinator considers all active transactions (i.e., transactions that have started 2PC but without final decisions in the stable log) at the time of the failure as aborted ones. The *presumed abort* protocol (PrA) is designed to reduce the cost associated with aborting transactions by making the abort presumption of PrN explicit (Mohan et. al., 1986).

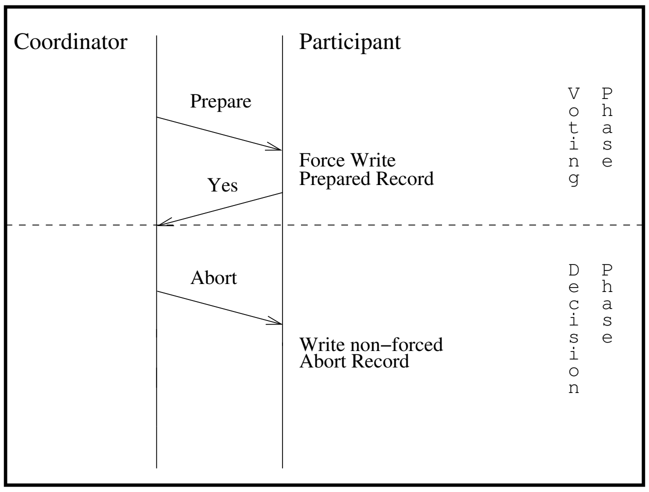
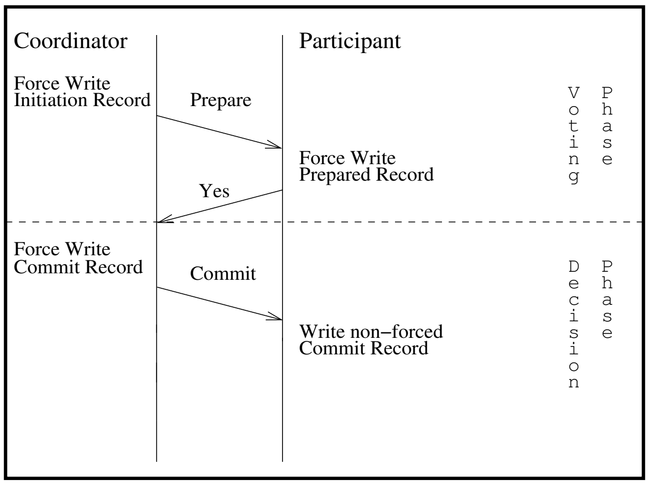


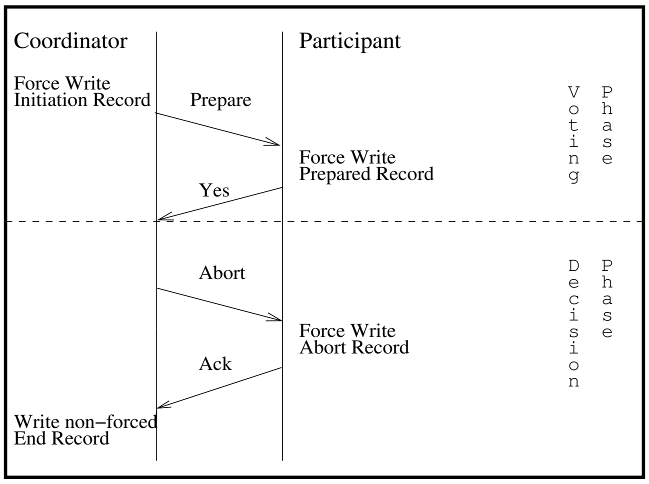
Figure 5: The presumed abort (PrA) protocol.

Specifically, in PrA, when a coordinator decides to abort a transaction, it does not force write the abort decision in its log as in PrN (see Figure 5). It just sends abort messages to all the participants that have voted "yes" and discards all information about the transaction from its protocol table. That is, the coordinator of an aborted transaction does not have to write any log records or wait for acknowledgments. Since the participants do not have to acknowledge abort decisions, they are also not required to force write such decisions. After a coordinator or a participant failure, if the participant *inquires* about a transaction that has been aborted, the coordinator, not remembering the transaction, will direct the participant to abort it (by presumption). On the other hand, the case of a commit decision remains the same as in PrN.

As opposed to PrA, the *presumed commit* protocol (PrC) is designed to reduce the cost of committing transactions (Mohan et. al., 1986). Instead of interpreting missing information about transactions as abort decisions, in PrC, coordinators interpret missing information about transactions as commit decisions. However, in PrC, a coordinator has to force write an *initiation* record (which is also called *collecting* in (Mohan et. al., 1986)) for each transaction before sending 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 failure.



(a) Commit case.



(b) Abort case.

Figure 6: The presumed commit (PrC) protocol.

To commit a transaction (see Figure 6 (a)), the coordinator force writes a commit record to logically eliminate the initiation record of the transaction and then sends out the commit decision. 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. After a coordinator or a participant failure, if the participant inquires about a transaction that has been committed, the coordinator, not remembering the transaction, will direct the participant to commit it (by presumption).

To abort a transaction (see Figure 6 (b)), 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 the acknowledgments before discarding all information pertaining to the transaction. When a participant receives the decision, it force writes an abort record and then acknowledges the decision, as in PrN.

Table 1: The costs for update transactions in 2PC and its most commonly known two variants.

2PC Variant	Commit Decision						Abort Decision					
	Coordinator			Participant			Coordinator			Participant		
	m	n	p	m	n	q	m	n	p	m	n	q
Presumed Nothing	2	1	2	2	2	2	2	1	2	2	2	2
Presumed Abort	2	1	2	2	2	2	0	0	2	2	1	1
Presumed Commit	2	2	2	2	1	1	2	1	2	2	2	2

Table 1 summarizes the costs associated with the three 2PC variants for the commit as well as the abort case assuming a "yes" vote from each participant: '*m*' is the total number of log records, '*n*' is the number of forced log writes, '*p'* is the number of messages sent from the coordinator to each participant and '*q*' is the number of messages sent back from each participant to the coordinator.

Taxonomy of Atomic Commitment in MDBSs

Unlike (homogeneous) distributed database systems, the constituent LDBMSs in an MDBS (or the Internet) might use different atomic commit protocols such as the basic 2PC protocol, that we discussed above, or one of its variants. Furthermore, some LDBMSs might be legacy systems (such as centralized database systems or file-based databases) that do not support any form of ACPs. For this reason, a LDBMS can be classified as either *externalized* or *non-externalized* LDBMS. An externalized LDBMS: (1) implements an ACP and (2) makes the system calls pertaining to its commit protocol, called *commit operators*, available to the outside world through its interface. Otherwise, a LDBMS is called non-externalized.

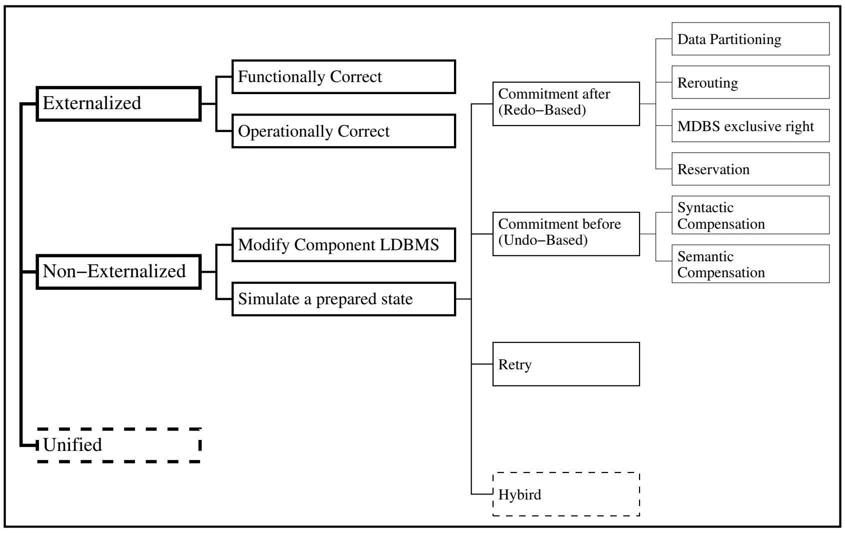


Figure 7: Taxonomy of schemes for atomic commitment in heterogeneous database environments.

Figure 7 depicts three main categories of approaches that ensure the atomicity of global transactions in heterogeneous database environments. These three categories represent the first level of classification in our taxonomy. The first two categories are based on the assumptions that the different schemes make about the externalization of ACPs by the underlying LDBMSs that we mentioned above (externalized vs. non-externalized), whereas the third approach is derived from the other two approaches through interoperating them in the same environment and is called *unified*. In what follows, we present the details of our taxonomy through further refinements to the main three categories and through associating each of the proposed schemes, as it appears in the literature, with the appropriate category in our classification to the schemes that ensure atomicity in heterogeneous database environments. Thus, providing a way to understand the relationships as well as the differences among the different schemes.

Externalized Approach

Given the current distributed transaction processing standards (X/Open, 1996; ISO, 1998) and the continuous standardization efforts in distributed transaction processing, the research in this direction is based on the assumption that future LDBMSs will support ACPs with externalized commit operators. Thus, the challenge is to integrate LDBMSs that use different and incompatible ACPs. The incompatibility of ACPs means that the semantics of the coordination messages and the actions that are taken by a LDBMS that uses one ACP might be completely different than their counterparts in another ACP. Thus, integrating LDBMSs that use incompatible ACPs in an MDBS is not a trivial task as it was previously believed (Breitbart et. al., 1992; Pu et. al., 1991; Tal and Alonso, 1994). That is, it is not simply the case that once a LDBMS supports an externalized ACP, it can be integrated in an MDBS regardless of the ACPs used by the other LDBMSs in a straight forward manner (Gligor and Lunckenaugh, 1984; Skeen, 1981).

Some researchers have concentrated in resolving the incompatibility of ACPs with respect to the semantics of the coordination messages. Hence, this group of researchers was interested in achieving *functional correctness* of ACPs according to our definitions. That is, only achieving atomicity of global transactions without any regard to the practicality of the resulting integration of ACPs. Other researchers looked at the atomicity of global transactions from a more pragmatic point of view. That is, achieving *operational correctness*, according to our definitions, in which the outcome of terminated transactions should, eventually, be forgotten without sacrificing consistency. In the following two sections, we overview the research in both directions.

- Functionally Correct Integration of ACPs

Pu et. al., (1991) concentrated on integrating *asymmetric* and *symmetric* classes of ACPs in the Harmony prototype. Harmony integrates *centralized* LDBMSs where each externalizes one of the two classes of ACPs. In asymmetric protocols, such as 2PC, only the coordinator is responsible about making the final decision pertaining to a transaction whereas in symmetric protocols, such as *decentralized* 2PC (Skeen, 1981), all participating sites have the same right in the execution of the protocol and can commit the transaction independently.

In the Harmony project, a component system called *Supernov*a is responsible for the translation of the semantics of the coordination messages of the protocols used by the different LDBMSs. In the event that some LDBMSs use asymmetric commit protocols while the others employ symmetric ones in a transaction’s execution, Supernova is the coordinator of all asymmetric ACPs and a member of the symmetric ones. To ensure the atomicity of a transaction, Supernova does not send out its vote to the symmetric members until it hears from all LDBMSs that use asymmetric ACPs. Thus, the symmetric participants are prohibited from making a unilateral decision that might jeopardize atomicity until they receive the vote of Supernova.

Unlike the previous work which assumes that each database site is centralized with externalized ACP, Tal and Alonso (1994) assume that each LDBMS is a *distributed* database system that externalizes an ACP. In this case, ensuring the atomicity of transactions is further complicated because a LDBMS, being distributed, might reach a different decision about the outcome of a transaction than the other participating LDBMSs in the transaction's execution. For example, if the GTM sends out prepare to commit messages to the LDBMSs participating in a global transaction's execution, the LDBMSs might reach different decisions about the outcome of the transaction if each one of them uses decentralized 2PC. This is because a prepare to commit message in decentralized 2PC has a dual role (i.e., it tells each LDBMS that the transaction has finished its execution and at the same time the GTM's vote). Thus, one LDBMS might commit the transaction if all the participants at its site have exchanged "yes" votes while another LDBMS might abort the same transaction if any participant at its site has decided to abort (i.e., voted "no").

By using *auxiliary* participant processes at each LDBMS, Tal and Alonso (1994) interoperate PrN, *linear 2PC* (L2PC), *decentralized 2PC* (D2PC) and *three-phase commit* (3PC) protocols. For example, in the case of a D2PC LDBMS, the auxiliary participant, which can be thought of as the agent in the MDBS model, is used to prohibit the other participants at its site from making a decision by not sending its vote to the participants at its site until it receives the final decision from the GTM. In this way, the participants are blocked from being able to make a commit decision that might latter jeopardize the atomicity of the transaction. Once the auxiliary participant receives the GTM decision, it propagates the decision to the other participants and completes the protocol.

- Operationally Correct Integration of ACPs

The above two research works achieve only functional correctness. That is, reaching consistent decisions regarding the outcome of global transactions. In this section, we overview the research that achieve operational correctness.

Mohan et. al., (1993) proposed a new 2PC variant called the *generalized presumed abort* protocol (GPA). GPA is a modified version of the *presumed abort* protocol (PrA). The motivation behind this protocol is to achieve the best of the two worlds namely, the generality of the *peer-to-peer* and the performance of the *client-server* architectures, when considering ACPs. To allow for smooth migration from IBM's PrN, which is a 2PC variant that is adopted in the IBM SNA LU6.2 architecture, to IBM's GPA, the GPA protocol had to be a superset of the IBM's PrN protocol. Furthermore, the designers of GPA have concentrated on the implementation aspects of PrA while taking into considerations some of the issues that are specific to the SNA LU6.2 architecture, such as supporting *heuristic decisions*. Hence, PrA had to be modified in order to fit into SNA's architecture to achieve operational correctness.

Al-Houmaily and Chrysanthis (1999) explicitly defined operational correctness for *integrated ACPs* and showed the difficulties in achieving it even when interoperating the simplest and most closely related ACPs. Specifically, they used PrN, PrA and PrC to demonstrate these difficulties and proposed *presumed any* (PrAny) protocol that interoperates the three 2PC variants according to the operational correctness criterion and without modifying any of the original 2PC variants. They accomplished this in PrAny by recognizing and defining the notion of *safe state*, a state that determines the conditions under which a coordinator is able to forget terminated (i.e., either committed or aborted) transactions without sacrificing the consistency of its decisions when replying to the inquires of the participants after a failure.

Al-Houmaily (2008) systematically analyzes the different sources of incompatibilities among ACPs. Based on his analysis to incompatibilities, he proposed *Integrated Two-Phase Commit* (I-2PC) protocol that integrates *Implisit Yes-Vote* (IYV) (Al-Houmaily and Chrysanthis, 2000), which is a one-phase commit protocol, besides the three basic 2PC variants (i.e., PrN, PrA and PrC). He also concluded that incompatibilities could be due to (1) the semantics of the coordination messages (which include both their meanings as well as their existence), or (2) the presumptions that are made about the outcome of forgotten (terminated) transactions by the different ACPs in case of failures.

In the next section, we explain the notion of the safe state by discussing the basic idea behind PrAny.

- - The Presumed Any (PrAny) Protocol

As mentioned above, PrAny interoperates the three commonly known 2PC variants, namely PrN, PrA and PrC, in spite of their incompatibilities. The three protocols are incompatible not only because of the differences in the semantics of coordination messages (i.e., their absence from one variant vs. their presence in another one), but more importantly because of the contradicting presumptions that they make about the outcome of forgotten terminated transactions in case of failures. Recall that PrN does not make any (explicit) presumption about terminated transactions, PrA presumes abort, and PrC presumes commit .

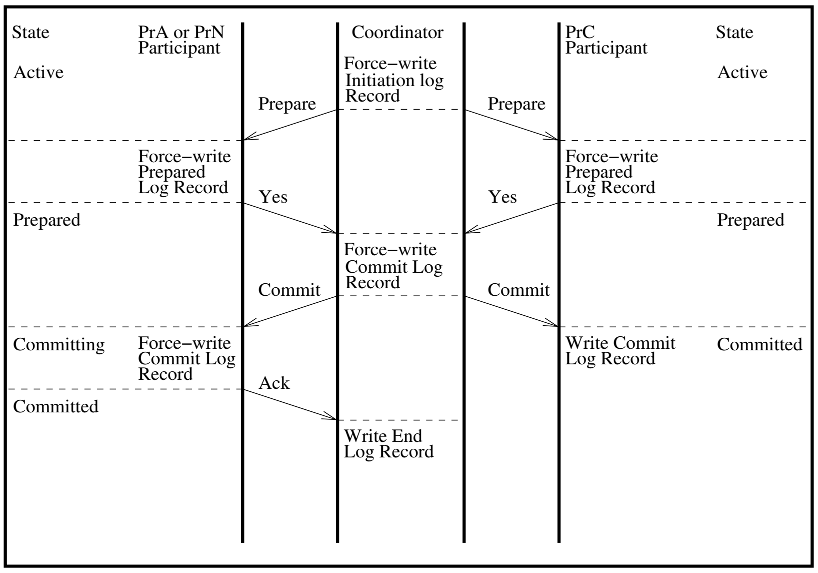
PrAny resolves the incompatibilities among the three variants by (1) "talking" the language that each protocol understands with respect to messages and (2) synchronizing the timing at which it forgets the outcome of terminated transactions. To illustrate the operation of PrAny, assume that there are three participants in a transaction's execution and the GTM (which is the coordinator of all (distributed) global transactions) is using PrAny. Furthermore, assume that one participant is using PrN, another participant is using PrA and the third participant is using PrC.

In PrAny, the GTM records the 2PC protocol employed by each participant in a table called *participants' commit protocol* (PCP). The PCP is kept onto stable storage and is updated when a new site joins or leaves the MDBS. Only a portion of the PCP, called *active participants' protocols* (APP) table, is maintained in main memory, containing the identities (IDs) of the participants with active transactions.

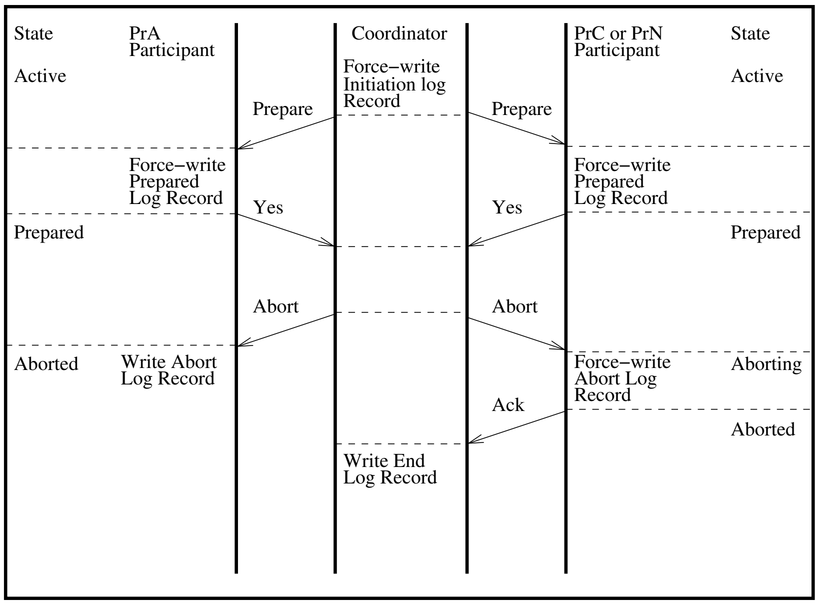
At commit time of a global transaction, the GTM refers to its APP to decide which protocol to use with the participants in the execution of the transaction. The GTM either selects PrN if all the participants are using PrN, PrA if all the participants are using PrA, or PrC if all the participants are using PrC.

In the event that some of the participants are using PrA while the others are using PrN or PrC, the GTM selects PrAny. From the GTM's perspective, PrAny consists of the same two phases, i.e., the voting phase and the decision phase, as in PrN, PrA and PrC (see Figure 8). The only distinction between PrAny and the other variants is in (1) the logging activities at the coordinator's site and (2) the timing at which the coordinator can safely forget about the outcome of transactions (i.e., reaching a safe state).

In PrAny, the GTM starts the voting phase by force writing an *initiation* record which includes the identities of the participants, as in PrC. However, unlike PrC, the initiation record also includes the protocol used by each participant. Then, the GTM sends to each participant a prepare to commit message. Once the GTM receives the votes from all the participants, it force writes a commit record if the decision is commit (see Figure 8 (a)). If the decision is abort, no decision record is written into the log (see Figure 8 (b)). Then, the GTM sends its final decision to all the participants. On a commit final decision, the GTM writes a non-forced end record once all the PrN and PrA participants acknowledge the decision. On an abort final decision, on the other hand, the GTM writes an end record once all the PrN and PrC participants acknowledge the decision. After that, the GTM forgets the transaction by discarding all information pertaining to the transaction from its protocol table.



(a) Commit case.



(b) Abort case.

Figure 8: The presumed any (PrAny) protocol.

The recovery procedure in case of communication and participants' failures are handled in a manner similar to the way they are handled in PrN, PrA and PrC protocols. According to the behavior of PrN, PrA and PrC, the GTM expects those participants that employ PrN and PrA to acknowledge commit final decisions but not those participants that employ PrC (see Figure 8 (a)). Thus, the GTM forgets the outcome of a committed transaction once the PrN and PrA participants acknowledge the commit decision, knowing that only a participant that employs PrC might inquire about the decision in the future. If a PrC participant inquires about a (commit) final decision after the GTM has forgotten the transaction, the GTM, knowing that the participant is using PrC, will direct the participant to commit the transaction, according to the presumption used in PrC.

Similarly, if the GTM makes an abort final decision, it expects only those participants that employ PrN and PrC to acknowledge the decision but not those employing PrA (see Figure 8 (b)). Hence, the GTM forgets about the outcome of an aborted transaction once the PrN and PrC participants acknowledge the abort decision. If a PrA participant inquires about an (abort) final decision after the GTM has forgotten the transaction, the GTM, knowing that the participant is using PrA, will direct the participant to abort the transaction, according to the presumption used in PrA.

Thus, besides talking the languages used by the different participants and synchronizing the timing at which the GTM forgets the outcome of terminated transaction, PrAny utilizes the knowledge of the GTM about the used protocols by the different participants. That is, when a participant inquires about a transaction that has been forgotten by the GTM, unlike the other 2PC variants, the GTM does not make any presumption in its own. Instead, it uses the presumption of the protocol used by the participant. For a PrC, the response of the GTM is commit whereas, for a PrA participant, the response of the GTM is abort. For a PrN participant, on the other, it is not possible that such a participant will inquire about the outcome of a transaction after it has been forgotten by the GTM. This is because both commit and abort decisions are acknowledged by PrN participants. By using such a presumption strategy, the GTM is always in a safe state with respect to the outcome of forgotten terminated transactions.

For the recovery of the GTM after a site failure, the GTM re-builds its protocol table by analyzing its stable log. For each transaction that has a decision log record without an initiation record, it means that PrN or PrA has been used for its commitment. For each such transaction without an end record, the GTM adds the transaction into its protocol table and re-initiates the decision phase with the recorded decision. In the case of PrA, the decision is always commit since PrA requires only commit decisions to be recorded in the log. In the case of PrN, the decision could be either commit or abort.

For each transaction that has an initiation record, it means that PrC or PrAny has been used for its commitment. Depending on the identities of the participants recorded in the initiation record and the protocols that they use, the GTM determines which of the protocols was used for the commitment of the transaction. For each transaction that PrC was used for its commitment and has no commit or end log record, the GTM adds the transaction into its protocol table and re-initiates the decision phase with an abort decision in accordance to PrC protocol.

Finally, for each transaction that PrAny was used for its commitment and has only an initiation record, or has initiation and commit records but without a corresponding end record, the GTM adds the transaction into its protocol table. In the former case, since either no decision was made or abort was decided before the failure, the GTM submits an abort decision to the PrN and PrC participants. The GTM does not include the PrA participants in accordance to PrA protocol [2]. In the latter case, since a commit decision record is found, the GTM submits a commit decision to the PrN and PrA participants but, in accordance to PrC protocol, not to PrC participants.

As during normal processing, after sending out a decision, the GTM waits for acknowledgments from PrN and PrC participants in the case of an abort decision and from PrN and PrA participants in the case of a commit decision. When a participant receives a final decision, it enforces and acknowledges the decision if it has not already enforced the decision prior to the failure. Otherwise, the participant simply acknowledges the decision [3]. When all the expected acknowledgments arrive, the GTM writes an end log record and forgets the transaction.

Non-Externalized Approach

In the previous section we discussed the methods that interoperate externalized LDBMSs. However, most of the literature about MDBSs consider legacy systems and is based on the assumption that each LDBMS does not externalize an ACP. Thus, the challenge is to ensure the atomicity of global transactions in spite of the fact that each LDBMS does not externalize an ACP. The approaches reported in the literature can be further classified into two categories, as shown (previously) in Figure 7. The first category of protocols suggests modifying component LDBMSs to support an externalized ACP while the second category of protocols achieves the atomicity of global transactions by *simulating* a prepared to commit state. In this section, we overview the methods that characterize both categories.

- Modify Component LDBMS

The researchers in the area of atomic commitment in MDBSs have realized the difficulties of ensuring the atomicity of global transactions and some have proposed the modification of component LDBMSs to support externalized ACPs (Gligor and Lunckenaugh, 1994; Pu et. al., 1991). These proposals do not only violate the autonomy requirement of LDBMSs; but might be impossible to achieve. This is because even if the man power required to make such modifications is available, the source code of LDBMSs is usually copyrighted and might not be available for such modifications. Hence, this solution is, generally, not practical.

- Simulate a Prepared State

The other alternative that ensures the atomicity of global transactions is to simulate a prepared to commit state at each LDBMS site without having to modify the source code of the LDBMS. The protocols proposed in this direction still violate the autonomy of the database sites with variant degrees, as shown by Chrysanthis and Ramamritham (1994), but less than the one that suggests modifying the source code of the component LDBMSs discussed above.

Simulating a prepared to commit state can be achieved using one of four approaches. The first two approaches are classified based on the relative commitment of the subtransactions pertaining to a global transaction at the participating LDBMSs with respect to the commitment of the global transaction by the GTM (Muth and Rakow, 1991). That is, in the first approach called *commitment after*, the

LDBMSs participating in a global transaction's execution commit the transaction locally after the GTM has made the final commit decision. Thus, in the event that a LDBMS aborts the transaction after the final commit decision is made but before the LDBMS has received the decision, the effects of the transaction should be *redoable*. In the second approach called *commitment before*, LDBMSs commit a transaction before the GTM has made its final commit decision. Thus, in the event that the GTM has finally decided to abort the transaction, the effects of the transaction should be *undoable*. In the third approach, the effects of a global transaction are always *retriable*. That is, if a LDBMS aborts a global transaction locally, the GTM should be able to re-execute the transaction at the LDBMS repeatedly until the transaction is finally committed at the LDBMS. The fourth approach is called *hybrid* because it is basically a derivation that combines the other three approaches.

- - Commitment After (Redo-based) Approach

In this approach, the prepared to commit state is simulated by characterizing the *denied local updates* (DLU) concept (Veijalainen and Wolski, 1992; Wolski and Veijalainen, 1991). When a subtransaction enters the simulated prepared to commit state, the DLU requires that no local transaction should be able to modify the states of the data objects accessed by the subtransaction. In this way, if the LDBMS at the site where the subtransaction is in its prepared to commit state has decided to abort the transaction unilaterally for any reason, the subtransaction can be redone and produce the same effects that it has produced just before it has been aborted by resubmitting it again.

In the literature, there are four ways where the DLU concept is characterized. The first one is based on data partitioning. In this method, the type of a transaction determines which type of data the transaction can read and which type of data it can modify (Breitbart and Silberschatz, 1992; Breitbart et. al., 1990; Breitbart et. al., 1992; Mehrotra et. al., 1992). For example, *globally updateable* data items can be only modified by global transactions whereas *locally updateable* data items can be only modified by local transactions. In the event that a LDBMS decides to abort a subtransaction and the final decision of the GTM is to commit the global transaction, the write operations of the subtransaction are redone by resubmitting a redo transaction that contains *only* the write operations of the subtransaction.

The second way is based on re-routing local transactions such that they have to go through the global transaction manager (or the agent at the site where they have been initiated) (Muth and Rakow, 1991; Soparkar et. al., 1991). Thus, the MDBS has complete control over local transactions and can abort any of them if DLU is violated.

The third way is the MDBS exclusive right (Barker and Özsu, 1991; Georgakopoulos, 1991 (a); Georgakopoulos, 1991 (b); Wolski and Veijalainen, 1991). In the MDBS exclusive right, no local transaction is allowed to start executing at a LDBMS after a site failure until the MDBSs has recovered all transactions that were in their simulated prepared to commit states. After a failure, the agent of the MDBS is guaranteed exclusive access by the LDBMS administrators until it recovers all global transactions, denying any local transaction request to access the database system.

The fourth way is *reservations* (Mullen, 1993; Mullen et. al., 1993). Reservations have two flavors. The first one called *generic* which is general enough to be used irrespective of the semantics of transactions and data while the other one is *semantics-based.* Both types of reservations characterize DLU by associating each pre-existing data item at each database site with a new one and modifying local transactions. In reservations, each subtransaction of a global transaction is associated with a reservation transaction. The reservation transactions of a global transaction are executed first. Only when all reservation transactions commits, the corresponding global transaction is executed. Otherwise, an unreservation transaction is executed for each committed reservation transaction.

In generic reservations, the extra data items serve as a form of high level locking to ensure the recoverability of global transactions by forcing them to conflict with local ones. This method is similar to the *ticket* method (Georgakopoulos et. al., 1991) that has been proposed to ensure serializability in MDBSs. When a reservation transaction executes, it marks each extra data item that is associated with the data item that will be accessed by the global transaction as “blocked”. If a reservation transaction encounters a “blocked” data item, it aborts and an unreservation transaction is executed for each successfully committed reservation transaction to “un-block” the data items that have been blocked by the corresponding reservation transaction. Local transactions are modified such that they abort if they attempt to modify a reserved data item (i.e., by checking its associated extra data item with respect to blocking). In this way, when the reservation subtransactions pertaining to a global transaction commits, the global transaction is guaranteed to commit once it starts its execution and despite failures. If a failure occurs, the failed subtransactions are redone and their effects will be the same as if no failure has occurred because the DLU is preserved. Once a global transaction is committed, an unreservation transaction is executed to "unblock" the data items that the transaction has accessed, allowing other local transactions and reservations to access these data items.

The semantics-based reservations exploit the semantics of transactions and data and are similar in principle to the *escrow* method proposed by O'Neil (1986) to enhance concurrency among transactions. However, semantics-based reservations have been proposed to ensure the atomicity of global transactions. In semantics reservations, a reservation transaction modifies the extra data items such that no local transaction can cause the global transaction to abort after its associated reservation has committed. Notice that, unlike generic reservations, local transactions are allowed to access reserved data items. For example, if a subtransaction needs to reserve five seats on a plane, its associated reservation subtransaction will read the "available number of seats" data item to check the availability of the requested number of seats. Then, the reservation transaction will set the extra data item to five if these seats are available. Local transactions, in this example, are modified such that they abort if they attempt to modify the "available number of seats" data item below the value recorded in the extra data item.

As we mentioned above, semantics-based reservations are similar in principle to escrow. Unlike escrow, however, transactions in both types of reservations execute on original data items but not the extra (or reserved) items which is the case in the escrow method.

- - Commitment Before (Undo-based) Approach
The atomicity of global transactions in the *commitment before* approach can be achieved in either a *syntactic* or *semantic* base. In the syntactic-based approach, the state of the database after the effects of a transaction has been rolled back is exactly the same as if the transaction had never modified any data items, which is the traditional notion of

atomicity. On the other hand, the semantics-based approach uses a weaker notion of atomicity called *semantics atomicity* in which the state of the data objects after a transaction has been rolled back does not necessarily correspond to their states as they were before the transaction had modified them.

Consider, for example, an airline reservation application. Using the traditional notion of atomicity, when a reservation subtransaction aborts, the "available number of seats" on the plane is exactly the same as it was before the subtransaction has started. In semantics-atomicity, a reservation subtransaction is allowed to commit and if the whole global transaction is to be finally aborted, a *compensating* subtransaction is executed. Since other transactions might execute between the commitment of the reservation subtransaction and its associated compensating subtransaction, the "available number of seats" data item might not be the same as it was before the reservation subtransaction had started.

To ensure syntactic compensation, no transaction should be able to access the data items modified by a transaction that is in its simulated prepared to commit state. This constraint can be achieved through re-routing of local transactions such that they have to access LDBMSs through the GTM (or the agents) (Muth and Rakow, 1991; Perrizo, 1991), as in the analogous method in the commitment after approach. In this way, the GTM has complete control over local transactions and can prohibit any of them from accessing data modified by a prepared to commit transaction. Furthermore, the GTM has to be able to execute an inverse operation for each of the operations of an aborted transaction such that the states of the data items modified by a transaction that has been locally committed but finally aborted by the GTM are exactly the same after executing the inverse operations. Thus, the GTM has to log the before images of the data items accessed by a prepared to commit global transaction. In this way, undoing the effects of an aborted global transaction is simply achieved by installing the before images of the data items that the transaction has modified (Perrizo, 1991).

On the other hand, semantic compensation can be achieved in two ways. In the first way, the subtransactions of a global transaction are *optimistically* committed in an individual manner. If a subtransaction is aborted, the GTM aborts the whole transaction. Therefore, a log is maintained in order to execute compensating transactions for the committed subtransactions. To ensure that a compensating transaction will eventually commit, no local transaction is allowed to execute between a subtransaction and its inverse compensating transaction. This is achieved by re-routing local transaction through the GTM which is the essence of the *optimistic commit protocol* (Levy et. al., 1991). Compensating transactions have been initially introduced by Gray (Gray, 1978) and used as the basis for the Sagas (Garcia-Molina and Salem, 1987) while Korth (Korth et. al., 1990) studied the formal aspects of compensating transactions.

The second way of semantics compensation is through reservations. Reservations permits more interleaving between local and global transactions and at the same time work in situations where the classical compensation fails (Mullen, 1993; Mullen et. al., 1993).

- - Retry Approach
The retry approach is based on the assumption that a transaction will never get involved in a consistency violation when re-executed after a failure (Hsu and Silberschatz, 1991). For example, in a banking system, a credit transaction will never violate consistency if there are no limits on the amounts of credits. Thus, such transactions will eventually commit if re-executed repeatedly. However, this approach has its limited applicability because transactions, in general, are not retriable. Also, notice that this approach differs from the redo approach since the whole transaction is executed again (including its read operations) but not only its write operations (which is the case in the redo approach). Thus, in the redo approach, a transaction will move the database to a state that is equivalent to the sate had the transaction committed in its first execution whereas in the retry approach, this is not necessarily the case.

- - Hybrid Approach
The hybrid approach is derived from the other three approaches discussed above, namely commitment after, commitment before and retry. For example, in the *pivot* method (Mehrotra et. al., 1992), a global transaction consists of a *pivot* subtransaction and a set of subtransactions where each of which is either compensatable of retriable. The pivot subtransaction is neither compensatable nor retriable. Furthermore, it is assumed that there is no value dependencies among the subtransactions (i.e., a write operation pertaining to a subtransaction is not a function of a read operation pertaining to

another subtransaction). To ensure the atomicity of such a transaction, the compensatable subtransactions are executed first and committed. Then, the pivot is executed and committed. Finally, the retriable subtransactions are executed and committed. When the pivot is committed, the whole transaction has to commit. Otherwise, the whole transaction has to abort. Since each subtransaction before the pivot is compensatable, a compensating transaction for each of these transactions is launched if the pivot aborts. On the other hand, if the pivot commits, the retriable subtransactions are repeatedly executed until they all commit. A more general hybrid method that combines the undo, redo and retry approaches has been also proposed by Breitbart (Breitbart et. al., 1992).

Unified Approach

The externalized and non-externalized approaches that we discussed thus far complement each other. That is, a method that can be applied to one LDBMS might not be applicable to another in an MDBS. The unified approach is derived by interoperating the methods of the other two approaches in the same environment. This direction of research was the focus of both Nodine (Nodine, 1993) and Mullen (Mullen, 1993). They both proposed unifying approaches in which they integrate the different methods that ensure the commitment of global transactions with minor differences with respect to their classifications and terminologies. Both works attempt to provide the most general and flexible framework that ensures the atomicity of transactions in spite of the diversity of the semantics of transactions and data.

In Mullen's work, the subtransactions pertaining to a global transaction are classified into *nine* categories as follows (Mullen, 1993):

1. *Implicitly Compensatable*: a subtransaction that is considered compensated without executing a compensating subtransaction on its behave (e.g., read-only transactions are considered implicitly compensatable because they do not update any data items).
2. *Compensatable*: a subtransaction that can be undone by executing an explicit compensating transaction.
3. *Reservable Compensatable:* a subtransaction that requires the execution of a reservation transaction before executing it. This is in order to be able to undo its effects when it fails by executing an un-reservation transaction.

4. *Preperable*: a subtransaction that can be prepared by entering a prepared-to-commit state at the executing site.
5. *Implicitly Reservable*: a subtransaction that is guaranteed to commit when it is redone without requiring a reservation subtransaction.
6. *Value Preserving Implicitly Reservable*: a subtransaction that is guaranteed to commit when redone without requiring any reservation transaction. In contrast to implicitly reservable subtransaction, this type of subtransactions requires that the read set of the subtransaction remains the same for the different executions of the subtransaction until it commits.
7. *Reservable*: a subtransaction that is guaranteed to commit when redone but requires the execution of a reservation transaction.
8. *Value Preserving Reservable*: a subtransaction that is guaranteed to commit when redone but requires the execution of a reservation transaction and that the read set of the subtransaction remains the same for the different executions of the subtransaction until it commits.
9. *Non-Compensatable-Preperable-Reservable*: a subtransaction that is neither compensatable nor reservable, explicitly or implicitly, and is not preperable either.

On the other hand, in Nodine's work, the subtransactions pertaining to a global transaction are classified into *four* categories as follows (Nodine, 1993):

1. *Agreement Protocols*: are those that are used to ensure unanimous decision about the final outcome of a global transaction by all participating sites such as 2PC.
2. *Blocking Protocols*: are those that simulate a prepared-to-commit state by blocking other transactions from interfering with the commitment of a global transaction, such as data partitioning and reservations.
3. *Stratification Protocols*: are those that ensure semantic atomicity where each transaction can never observe the committed effects of another transaction at one site and its undone effects at another site.
4. *Serialization Protocols*: are those that ensure correct serialization order of retried subtransactions in the global serialization order.

Although both of the above works are similar in principle, we find that Mullen's unified approach is a more general framework. It thoroughly analyzes the different types of subtransactions pertaining to a global transaction and formally describes not only the class of transactions that is not committable, but also the class of transactions that are committable. Our work differs from Mullen's and Nodine's in two aspects. First, rather than analyzing the different possible types of subtransactions, we classify the different schemes that ensure atomicity as they appear in the literature. For example, in our taxonomy, the *pivot* method is classified as a hybrid method that is both compensatable and retriable. Second, our taxonomy deals with externalized approaches more thoroughly by distinguishing between *functionally* and *operationally* correct schemes of ACPs. Thus, we believe that our work is simpler and more comprehensive than previous works. This has the advantage of providing a better understanding to atomicity in heterogeneous database environments that should help in devising new and more flexible schemes.

Summary and Conclusion

Database transactions exhibit attractive consistency and reliability guarantees that make them very appealing basic building blocks for the development of advanced software application systems such as electronic commerce and electronic services, web-based transactions and multi-organizational workflows. As such, it is imperative to provide universal transactional support for these applications that access multiple and heterogeneous database sites and in particular, guaranteeing the atomicity of transactions.

For the above reason, this paper concentrated on the issue of *atomicity*. Specifically, it (1) thoroughly investigated the different schemes that have been proposed to characterize atomicity in heterogeneous database systems and (2) classified them in a form of taxonomy. The taxonomy is essentially based on the assumptions that the different approaches make about the mechanisms used for atomicity by the different database sites. We believe that such taxonomy is a necessary first step towards a better understanding to atomicity in heterogeneous database systems and a framework that can be used for the development of new and more flexible schemes for atomicity, an extremely important aspect for the implementation of reliable advanced Internet software application systems.

References

Abdallah, M., Guerraoui, R. and Pucheral, P. (2002) "Dictatorial Transaction Processing: Atomic Commitment without Veto Right", Distributed and Parallel Databases, Vol. 11, No. 3, pp. 239-268.

Al-Houmaily, Y. J. (1997) "Commit Processing in Distributed Database Systems and in Heterogeneous Multidatabase Systems," Ph.D. Thesis, Department of Electrical Engineering, University of Pittsburgh, USA.

Al-Houmaily, Y. J. (2005) "On Interoperating Incompatible Atomic Commit Protocols in Distributed Databases", Proceedings of the 1st IEEE International Conference on Computers, Communications, and Signal Processing, Kuala Lumpur, Malaysia.

Al-Houmaily, Y. J. (2008) "Incompatibility Dimensions and Integration of Atomic Commit Protocols", International Arab Journal of Information Technology, Vol. 5, No. 4, October 2008.

Al-Houmaily, Y. J. and Chrysanthis, P. K. (1999) "Atomicity with Incompatible Presumptions", Proceedings of the 18th ACM Symposium on Principles of Database Systems.

Al-Houmaily, Y. J. and Chrysanthis, P. K. (2004 (a)) "1-2PC: The One-Two Phase Atomic Commit Protocol", Proceedings of the ACM Annual Symposium on Applied Computing, Special track on database theory, technology and applications, Nicosia, Cyprus.

Al-Houmaily, Y. J. and Chrysanthis, P. K. (2004 (b)) "ML-1-2PC: An Adaptive Multi-level Atomic Commit Protocol", Proceedings of the 8th East European Conference on the Advances in Databases and Information Systems, Budapest, Hungary.

Al-Houmaily, Y. J. and Chrysanthis, P. K. (2000) "An Atomic Commit Protocol for Gigabit-Networked Distributed Database systems", Journal of Systems Architecture, Vol. 46, pp. 809-833.

Al-Houmaily, Y. J., Chrysanthis, P. K. and Levitan, S. P. (1997) "An Argument in Favor of the Presumed Commit Protocol", Proceedings of the 13th IEEE International Conference on Data Engineering, pp. 255-265.

Attalui, G. and Salem, K. (2002) "The Presumed-Either Two-Phase Commit Protocol", IEEE Transactions on Knowledge and Data Engineering, Vol. 14, No. 5, pp. 1190-1196.

Barga, R., Lomet, D., Shegalov, G. and Weikum, G. (2004) "Recovery Guarantees for Internet Applications", ACM Transactions on Internet Technology, Vol. 4, No. 3.

Barker, K. and Özsu, M. T. (1991) "Reliable Transaction Execution in Multidatabase Systems", Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems, pp. 344-347.

Breitbart, Y., Garcia-Molina, H. and Silberschatz, A. (1992) "Overview of Multidatabase Transaction Management", VLDB Journal, Vol. 1, No. 2, pp. 181-239.

Breitbart, Y. and Silberschatz, A. (1991) "Strong Recoverability in Multidatabase Systems", Proceedings of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, pp. 170-175.

Breitbart, Y., Silberschatz, A. and Thompson, G. (1990) "Reliable Transaction Management in a Multidatabase System", Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 215-224.

Breitbart, Y., Silberschatz, A. and Thompson, G. (1992) “Transaction Management Issues in a Failure-Prone Multidatabase System Environment”, VLDB Journal, Vol. 1, No. 1, pp. 1-39.

Chrysanthis, P. and Ramamritham, K. (1994) “Autonomy Requirements in Heterogeneous Distributed Database Systems”, Proceedings of the Conference on the Advances on Data Management, pp. 283-302.

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, Prentice Hall, pp. 370-416.

Garcia-Molina, H. and Salem, K. (1987) “Sagas”, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 249-259.

Georgakopoulos, D. (1991 (a)) “Transaction Management in Multidatabase Systems”, Ph.D. Thesis, Department of Computer Science, University of Houston, Houston, Texas, USA.

Georgakopoulos, D. (1991 (b)) “Multidatabase Recoverability and Recovery”, Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems, pp. 348-355.

Georgakopoulos, D., Rusinkiewics, M. and Sheth, A. (1991) “On Serializability of Multidatabase Transactions Through Forced Local Conflicts”, Proceedings of the 7th IEEE International Conference on Data Engineering, pp. 348-355.

Gligor, V. and Lunckenaugh, G. (1984) “Interconnecting Heterogeneous Database Management Systems”, IEEE Computer, Vol. 17, No. 1, pp. 33-43.

Gray, J. (1978) “Notes on Data Base Operating Systems”, in Bayer R., Graham, R. M. and Seegmuller, G. (Eds.), Operating Systems: An Advanced Course, Lecture Notes in Computer Science, Springer-Verlag, Vol. 60, pp. 393-481.

Gray, J. (1981) “The Transaction Concept: Virtues and Limitations”, Proceedings of the 7th International Conference on Very Large Databases, pp. 144-154.

Gupta, R., Haritsa, J. and Ramamritham, K. (1997) “Revisiting Commit Processing in Distributed Database Systems”, Proceedings of the ACM SIGMOD International Conference on the Management of Data.

Haerder, T. and Reuter, A. (1983) “Principles of Transaction-Oriented Database Recovery”, ACM Computing Surveys, Vol. 15, No. 4, pp. 287 - 317.

Hsu, M. and Silberschatz, A. (1991) “Unilateral Commit: A New Paradigm for Reliable Distributed Transaction Processing”, Proceedings of the 7th IEEE International Conference on Data Engineering, pp. 286-293.

ISO (1998), “Open Systems Interconnection – Distributed Transaction Processing - Part 1: OSI TP Model”, ISO/IEC 10026-1.

Korth, F., Levy, E. and Silberschatz, A. (1990) “A Formal Approach to Recovery by Compensating Transactions”, Proceedings of the 16th International Conference on Very Large Databases, pp. 95-106.

Lampson, B. (1981) “Atomic Transactions”, in Lampson (Ed.), Distributed Systems: Architecture and Implementation - An Advanced Course, Lecture Notes in Computer Science, Springer-Verlag, Vol. 105, pp. 246-265.

Lampson, B. and Lomet, D. (1993) “A New Presumed Commit Optimization for Two Phase Commit”, Proceedings of the 19th VLDB Conference, pp. 630-640.

Levy, E., Korth, H. and Silberschatz, A. (1991) “An Optimistic Commit Protocol for Distributed Transaction Management”, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 88-97.

Mehrotra, S., Rastogi, R., Breitbart, Y., Korth, H. and Silberschatz, A. (1992) “Ensuring Transaction Atomicity in Multidatabase Systems”, Proceedings of the ACM Symposium on Principles of Database Systems, pp. 164-175.

Mehrotra, A., Rastogi, R., Korth, H. and Silberschatz, A. (1992) “A Transaction Model for Multidatabase System”, Proceedings of the International Conference on Distributed Computing Systems, pp. 56-63.

Mohan, C., Britton, K., Citron, A. and Samaras, G. (1993) “Generalized Presumed Abort: Marrying Presumed Abort and SNA's LU 6.2 Commit Protocols”, Proceedings of the 5th International Workshop on High Performance Transaction Systems.

Mohan, C., Lindsay, B. and Obermarck, R. (1986) “Transaction Management in the R^* Distributed Data Base Management System”, ACM Transactions on Database Systems, Vol. 11, N. 4, pp. 378-396.

Mullen, J. (1993) “Atomic Commitment in Multidatabase Systems”, Ph.D. Thesis, Department of Computer Science, Purdue University, West Lafayette, Indiana, USA.

Mullen, J., Jing, J. and Sharif-Askary, J. (1993) “Reservation Commitment and its use in Multidatabase Systems”, Proceedings of the 4th IEEE International Conference on Database and Expert Systems Applications (DEXA), pp. 116-121.

Muth, P. and Rakow, T. (1991) “Atomic Commitment for Integrated Database Systems”, Proceedings of the 7th IEEE International Conference on Data Engineering, pp. 296-304.

Nodine, M. (1991) “Interactions: Multidatabase Support for Planning Applications”, Ph.D. Thesis, Department of Computer Science, Brown University, USA.

O’Neil, P. (1986) “The Escrow Transactional Method”, ACM Transactions on Database Systems, Vol. 11, No. 4.

Perrizo, W., Rajkumar, J. and Ram, P. (1991) “HYDRO: A Heterogeneous Distributed Database System”, Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 32-39.

Pu, C., Leff, A. and Chen, S. (1991) “Heterogeneous and Autonomous Transaction Processing”, IEEE Computer, Vol. 24, No. 12, pp. 64-72.

Samaras, G., Kyrou, G. and Chrysanthis, P. K. (2003) “Two-Phase Commit Processing with Restructured Commit Tree”, Proceedings of the Panhellenic Conference on Informatics, Lecture Notes on Computer Science, Vol. 2563, pp. 82-99.

Skeen D. (1981) “Non-blocking Commit Protocols”, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 133-142.

Soparkar, N., Korth, H. and Silberschatz, A. (1991) “Failure-Resilient Transaction Management in Multidatabases”, IEEE Computer, Vol. 24, No. 12, pp. 28-36.

Stamos, J. and Cristian, F. (1993) “Coordinator Log Transaction Execution Protocol”, Distributed and Parallel Databases, Vol. 1, pp. 383-408.

Tal, A. and Alonso, R. (1994) “Integration of Commit Protocols in Heterogeneous Databases”, Distributed and Parallel Databases, Vol. 2, No. 2, pp. 209-234.

Veijalainen, J. and Wolski, A. (1992) “Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases”, Proceedings of the 8th IEEE International Conference on Data Engineering, pp. 470-479.

Wolski, A. and Veijalainen, J. (1991) “2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase”, in Rishe, N., Navathe, S. and Tal, D. (Eds.), Databases: Theory, Design, and Applications, pp. 268-287.

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

Yu, W. and Pu, C. (2007) "A Dynamic Two-Phase Commit Protocol for Adaptive Composite Services", International Journal of Web Services Research, Vol. 4, No. 1.

Footnotes

[1] The same correctness criterion was defined in the context of integrated ACPs (Al-Houmaily, 1997; Al-Houmaily and Chrysanthis, 1999; Al-Houmaily 2008). Our definition here is a generalization that captures the correctness of *any* ACP rather than *only* integrated ACPs.

[2] A coordinator in PrA never re-submits an abort decision to the participants after its failure because it will not have any recollection about aborted transactions. It is the responsibility of the participants to inquire about the outcome of such transactions. Similarly, a coordinator in PrC never re-submits commit decisions to the participants after its failure.

[3] A participant without any memory regarding a transaction is assumed to have already received and enforced the decision and discarded all information pertaining to the transaction.