Commit Protocols for Externalized-Commit Heterogeneous Database Systems *

Ayellet Tal
Princeton University

Rafael Alonso
Matsushita Information Technology Laboratory

Abstract

In designing a heterogeneous database systems, one of the main technical challenges is developing techniques for ensuring global commit. That is, guaranteeing that a transaction spanning multiple individual database management systems (DBMSs) either commits at all the participating DBMSs or at none of them. Previous work in this area typically assumes that the participating DBMSs do not provide a mechanism for interacting with their commit facilities. While this is true in many cases, in practice there are systems which support a programmatic interface to their commit protocols. We refer to database systems offering such facilities as *externalized commit* DBMSs.

The focus of this paper is on commit protocols for these systems. We propose two new commit protocols for externalized commit DBMSs. The first may be used to obtain global commit in heterogeneous database systems composed of DBMSs with different 2-phase commit protocols (e.g., centralized and linear). The second protocol is more general, and ensures global commit even if the participating DBMSs employ 3-phase commit protocols. The more general protocol also preserves database autonomy, since it does not block a DBMS upon failure of another system. We describe both protocols in detail and prove their correctness.

Keywords: heterogeneous databases, commit protocols, distributed databases, autonomy, integration.

*This work was partially supported by an IBM Research Initiation Grant. A. Tal's address: Computer Science Department, Princeton University, Princeton, NJ 08544, (609) 258-1793, ayt@princeton.edu. R. Alonso's address: MITL, 2 Research Way, Princeton, NJ 08540, (609) 734-7328, alonso@mitl.research.panasonic.com

1 Introduction

Heterogeneous database systems allow not only the physical distribution of the data but also its logical distribution. There are many environments where a collection of autonomous databases which need to cooperate with each other already exists. These environments, and the problems involved in integrating a heterogeneous collection of databases have already been discussed in the literature (for example, see [8]). Although there has been much work in this area, many open problems remain. One of these open issues is support for atomic commit in heterogeneous database systems.

In considering commit protocols for heterogeneous database systems, we need to distinguish between two possible models of database system operation. The commit protocol for a database management system (DBMS) may be said to be either *externalized* or *non-externalized*. By *externalized* we mean that there exists a mechanism by which a process outside of the DBMS can take part in commit decisions, as well as recover after failures. Typically, such a mechanism is a library call interface. For example, users of Sybase's Open Client DB Library [17] may write an application program which sends a "PREPARE TRANSACTION" command to the database servers; user-coded Open Servers may inquire whether a particular transaction committed or not by issuing a *stat_xact()* call. [1]

Systems which do not provide such facilities are said to have a non-externalized commit protocol. Most previous work on heterogeneous commit protocols assumes the commit protocols of the individual DBMSs are of this type. We briefly survey previous work on non-externalized commit protocols in Section 2.

The focus of this paper is the development of global commit protocols for heterogeneous database systems composed of DBMSs which externalize their commit protocols. We believe the importance of externalized-commit systems will increase because they will become more prevalent. We argue that this will be so because, in the near future, centralized databases will be replaced by distributed ones, which usually externalize their commit protocols. Unfortunately, externalized commit databases have not been examined in detail in the past. Work in this area has consisted mostly of the various standardization efforts supported by industry (i.e., LU 6.2 [9], or OSI's TP service [19]). However, at this point it is not clear to us that any one standard will be chosen; furthermore, even if one becomes dominant in the database world, there are many other storage systems of interest (e.g., transactional file systems, mail repositories) that will very likely not follow the standard.

The main contributions of this paper are two novel heterogeneous commit

[1]For further detail see the "Two Phase Commit Example Program" in the Open Client DB Library Reference Manual for Sybase [17].

protocols for externalized commit database systems. The first may be used in an environment where the individual DBMSs have any combination of the following types of two-phase commit protocols (2PC): hierarchical, centralized, decentralized, or linear. The protocol depends on a set of assistant processes (*agents*) to ensure the correctness of the mechanisms. We also present a more general protocol that allows the integration of DBMSs having either two or three-phase commit protocols (3PC). This second protocol has a higher performance overhead than the first. However, in addition to its increased generality, it has the following advantage: any participating DBMS will not be stopped by the failure of any process external to that DBMS, thus preserving database autonomy. We prove the correctness of both protocols.

In the following section we present a partial overview of previous research in heterogeneous commit protocols and recovery techniques for non-externalized databases. In Section 3 we formally define our model. In Section 4 we describe how to obtain global commit in heterogeneous DBMSs using a variety of 2PC algorithms. In Section 5, we extend this result to include both 2PC and 3PC systems. We conclude by highlighting the results of this paper and discussing directions for future research. For the sake of completeness and consistency of terminology, in Appendix A we review the various homogeneous commit protocols to which we refer in this paper.

2 Non-Externalized Commit Databases

Many conventional databases do not make public their commit protocols and the concomitant states. The basic problem in merging the commit protocols of such databases is the unavailability of a visible *wait* state. Yet, we want to ensure atomicity and durability. In [11] it is shown that if autonomy of the local database systems is preserved, in general it is impossible to perform atomic commitment. In this section we outline some of the previously proposed approaches to this problem. As we will see below, if a database does not externalize its commit protocol, one must sacrifice functionality in order to construct a heterogeneous database system.

One obvious way to deal with the heterogeneous commit problem is to modify the databases. We can do this either by forcing all databases in the system to adopt a single commit protocol (converting the heterogeneous database into a conventional distributed database as far as commit goes), or by forcing local transactions to be managed by the global manager (which creates a bottleneck in the system). At any rate, modifying the databases is not always a feasible alternative.

In a recent survey of heterogeneous database transaction management [3] three categories of approaches are presented: *redo* (the writes of the failed subtransaction are reexecuted), *retry* (the entire subtransaction is run again) and *compensate* (the effects of a committed subtransaction are erased by executing a compensating transaction at a later time). We describe each of the three approaches below.

Most of the redo approaches have suggested limiting data access to enforce correct recovery. In [4] it is proposed to divide the data items into globally-updatable items and locally-updatable ones; moreover, transactions modifying globally-updatable data items would not be allowed to read locally-updatable items. A similar solution is used by [20] in their *2PC Agent Method* method. The drawback of this kind of approaches is that they restrict access to the objects in the databases.

The retry approach proposed in [12] involves re-running a failed subtransaction (if another one committed at a different site). However, as pointed out in [3], one needs to make sure that any values the original subtransaction reads were not passed to other subtransactions; it must also be true that the subtransaction is repeatable (i.e., if it is tried a sufficiently large number of times it will commit). Once again, these conditions may not be met in some environments.

For compensation to be an acceptable approach, one has to give up the demand for atomicity and choose a new correctness criterion. An example of this is the work on Sagas [5]. Clearly, compensating approaches are not generally applicable.

Yet another class of solutions to our problem lie in implementing 2PC by simulating the *wait* state. In [1] this is accomplished by modifying a global subtransaction so that it will send a message to the global transaction manager before committing. Moreover, the recovery mechanism is modified to give the global transaction manager exclusive access to the database after a site failure. There are several disadvantages to this approach: The databases must allow the execution of a remote application in the middle of a transaction, and it must be possible to grant exclusive access to a single user after a failure. The most severe problem is that the local database might be blocked if the global manager fails, and this has serious implications on autonomy. A related approach is presented in [6].

3 The Model

We assume our heterogeneous database is composed of individual homogeneous DBMSs, each of which externalizes its commit protocol. We assume that each DBMS uses one of the commit protocols described in Appendix A as its local commit protocol. No modifications are made to the participating database systems. Furthermore, none of them is ever aware of being part of a heterogeneous database system. We expect participating DBMSs to be distributed. We depict the architecture of the system in Figure 1. For example, participating distributed database DB1 in Figure 1 is composed of three processes, each potentially at a different site. We denote the three single-site database processes forming distributed DBMS DB1 by db11, db12, and db13.

With each participating DBMS we associate an agent (a process that will aid in the commit operation). As far as the DBMS processes are concerned, the agent is yet another database process (i.e., db11 views the agent for DB1 as being similar to say db12). There is a global manager that controls all the agents in the heterogeneous DBMS. The agents need not know about each other for the protocol in Section 4. They must have this information for the protocol studied in Section 5.

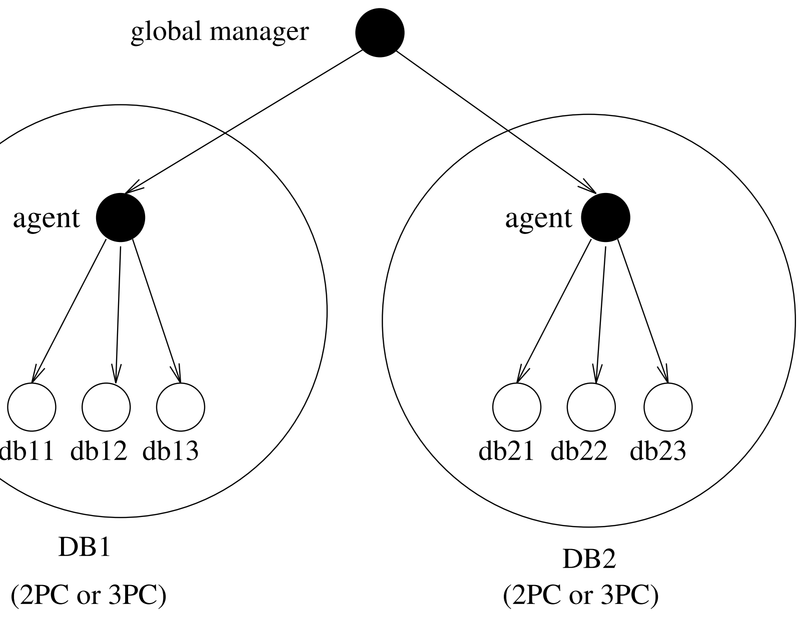

Figure 1: The Database Model

We define a *local* transaction as a transaction residing in a single distributed database. (Note that a local transaction may be executed in several sites if the participating DBMS is a distributed one.) The participating DBMSs process local transactions in their usual manner. Transactions spanning data in multiple individual DBMSs are called *global* transactions. Global transactions are submitted to the global manager, which in turn contacts the agents at the relevant DBMSs. Each agent starts a local transaction, and we assume that the agent becomes the transaction coordinator.

We expect each externalized commit DBMS to support two types of commands: those necessary for the commit processing (e.g., start transaction, vote to abort), as well as those required for recovery (e.g., a command to find out what commit decision was taken for a given transaction). For example, in the next two sections we assume that the transaction interface allows an EXEC statement (to start a local transaction), READ and WRITE operations, a PREPARE (3PC DBMSs only), ABORT, COMMIT, and a STATUS_REQ (request information about the commit status of a transaction).

The goal of the protocols described in the next two sections is to achieve a *global commit*. By this we mean that all DBMSs will either commit or abort a given global transaction. In Figure 2 we remind the reader of the five correctness requirements of an atomic commitment algorithm (see [2]).

1. All processes that reach a decision, reach the same one.
2. A process can not reverse its decision after it has reached one.
3. The commit decision can only be reached if all processes voted OK.
4. If there are no failures and all processes voted OK, the decision will be to commit.
5. Given any execution schedule containing failures (of the type that the algorithm is designed to tolerate), if all failures are repaired and no new failures occur for a sufficiently long time, then all processes will eventually reach a decision.

Figure 2: Correctness Criteria for Commit Protocols

As in homogeneous DBMS, there are several possible kinds of failures in heterogeneous DBMSs: transaction failures, system failures, media failures, process failure, site failures and communication failures. Our first protocol (Section 4) tolerates the same types of failures as 2PC protocols. The protocol presented in Section 5 assumes a reliable network and fail-stop site failures. In all cases we assume that, upon failure, local database processes behave in the manner prescribed by the local protocol (i.e., database processes follow the local termination protocol). See Appendix A for the description of 2PC and 3PC protocols and their recovery behaviors.

4 Merging 2PC Databases

In this section we consider a heterogeneous database composed of 2PC databases only. In practice, this is the most common case. We first describe how to integrate a variety of 2PC protocols. Next we comment on the integration of the recovery mechanisms. Finally, we prove our global protocol correct.

We start by presenting the global protocol. The global transaction manager runs a centralized 2PC protocol with the agents as participants. The global manager plays the role of coordinator. Below, we consider some versions of 2PC and describe how each local algorithm can be meshed with the global one.

Hierarchical: Adding a global transaction manager is similar to adding a new root to which all the local roots are connected. The agent waits for a message from the global transaction manager. When the agent gets the message, it forwards the message to the local processes. After collecting the answers from the local processes, the agent returns the answer to the global manager.

Centralized: The agent for each DBMS serves as the coordinator for the local transaction. When the global transaction manager sends its EXEC request, the agent sends the same request to the local database processes. If all the local processes reply OK to the agent, it then sends an OK to the global transaction manager. Otherwise the agent sends back a NOK. If the global transaction manager receives OK's from all the agents, it then tells all of them to COMMIT (or to ABORT otherwise). Each agent passes along the global message to the local processes.

Decentralized: The main difficulty in this case is that the initial message that the coordinator sends to the participants has two roles: it starts the commit process, and it also communicates to the others that the initiating process is willing to commit. Consider the following scenario: the global transaction manager asks its agent at the decentralized commit database to EXEC. If our agent now sends an OK to the local processes (OK and NOK are the only two kinds of messages it can send), the local processes will not only start the commit process, but also decide whether to commit or not. The agent can no longer enforce an ABORT decision if such a message were to be sent by the global transaction manager. A possible solution, shown in Figure 3, is to have an auxiliary process that collaborates with the agent. When the agent receives the EXEC message from the global coordinator, the agent sends a local OK to the local processes (including the auxiliary one). All the local processes exchange OK or NOK and the agent then knows whether the local database commits or not. The agent then sends OK or NOK to the global transaction manager. When the latter then sends an ABORT or COMMIT, the agent can tell the auxiliary to send NOK or OK to all the other local participants.

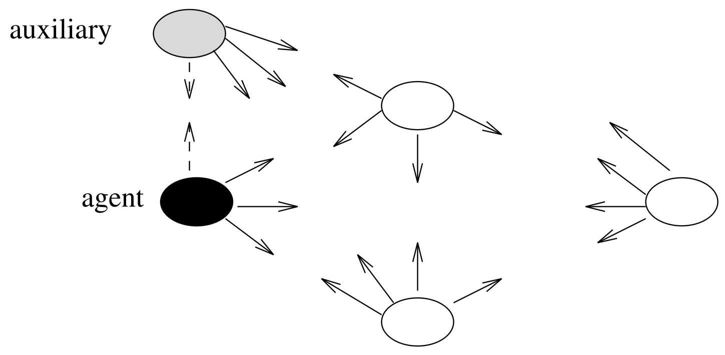

Figure 3: Decentralized 2PC

Linear: Here the COMMIT decision is made by the rightmost participant in the linear chain. We again need the help of an auxiliary process, which we force to be the rightmost participant in any commit involving a global transaction. (See Figure 4.) Our agent process starts the commit process; if an intermediate participant decides to abort, the auxiliary process eventually learns of it and tells the agent, who then informs the global manager. If all participants decide to commit, the auxiliary tells the agent, who then informs the global manager, and waits for the latter's reply, either a COMMIT or an ABORT; the agent then forces the auxiliary to send the appropriate response back down the chain of participants. There is an important point to be made here. In linear 2PC, the participants in the chain must have some knowledge regarding the next participant in the chain. In the above algorithm, we assume that the information is not pre-determined, but rather, that the process that starts the commit protocol specifies the linear order in its initiating OK command.

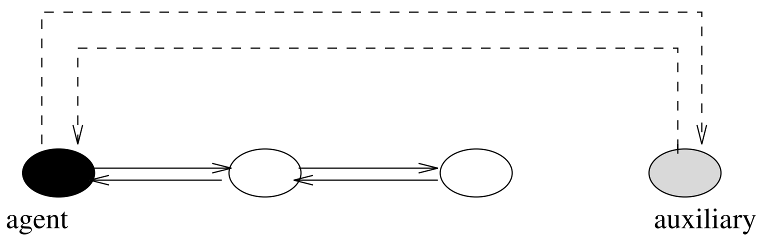

Figure 4: Linear 2PC

In [18] we formalize the protocol outlined above.

We now turn to the subject of recovery. We consider the recovery of database processes, agent processes, and that of the global manager. When a process recovers from a failure, it activates a termination protocol. If the process is a database one, it runs the local termination protocol (since it is not aware of the global transaction). However, when an agent recovers after a failure it must make sure that the local decision is consistent with the global one. The agent takes part in the both the local termination protocol and the global one (which is the termination protocol of a centralized 2PC database). Clearly, the agent needs to act as a participant in the global termination protocols, and as a coordinator in the local one. The agent first checks its log to find out its state before it failed. If it was in a *commit* state or in an *abort* state, it sends the local processes its decision. But, if the state was *wait*, the agent asks the global manager what action to take. The agent then passes the decision along to the local participants. A global manager recovering from a failure initiates the termination protocol of a coordinator in a *centralized* 2PC.

Theorem 1 *The algorithm presented above meets the correctness conditions 1-5 given in Figure 2.*

We prove this theorem by first showing that our modifications to the local commit protocols have not affected their correctness and then showing that the combined protocol is error-free.

We first show that our modifications do not affect the correctness of the local protocols. It is easy to see that this claim is true when the local 2PC is either hierarchical or centralized (because we only added a new process). However, for the cases of decentralized and linear 2PC, we have to show that the addition of the auxiliary process does not interfere with the correctness of the commit protocol. But in both linear and decentralized 2PC our auxiliary and agent processes act as just two more processes involved in the commit decision. The communication that goes on between them is transparent to the participants (other than perhaps a longer delay in the auxiliary's response), and thus cannot affect their decisions. And of course, agent and auxiliary obey all the rules of the protocol.

All that remains now is to justify the correctness of the global protocol. From the point of view of the global manager, the protocol is a hierarchical 2PC, since it is not aware of the internal variation of 2PC that each local database uses. The non-agent participants of the local databases are not aware of the hierarchy. The agents, however, are the only participants to have the knowledge that the protocol is neither a hierarchical 2PC nor any other variation. But, all the agent is doing (sometimes with the help of an auxiliary) is (1) getting instructions from the global manager and passing them to the local participants, and (2) collecting answers from the local participants and sending an answer to the global manager. This is exactly what an intermediate node in a hierarchical 2PC scheme does. Therefore, by a reasoning similar to that of the correctness proof for hierarchical 2PC we can show that our protocol is correct. □

5 Merging 2PC and 3PC Databases

In this section we consider the more general case of integrating databases using either 2PC or 3PC protocols. This case presents more of a challenge than the 2PC one, and results in a more interesting solution. As before, our global manager engages in a multi-phase protocol with the agents, and the latter integrate the local commit protocols into the global one.

This section is ordered as follows. We begin by suggesting why it might be impossible to use a global 2PC protocol between the global manager and the agents. We then consider the use of a 3PC protocol as the global one, and show that a straightforward use of 3PC is not adequate either. (This preliminary discussion of faulty protocols serves not only to emphasize the non-triviality of the problem but also sheds light on certain features of our protocol.) Finally, we present a new global protocol that does perform correctly; this protocol requires four phases and the use of auxiliary processes.

Consider using a 2PC algorithm between the global manager and the agents as the basis for the global commit protocol. As before, there is an agent process at each database (again, each database may be distributed among multiple sites); it is the agent's responsibility to communicate with the global transaction manager using the global commit algorithm, and to interact with the local processes using the local commit protocol. The messages between the global transaction manager and the agent responsible for a given 3PC database are: an EXEC from manager to agent, an OK or NOK as reply, and a COMMIT/ABORT decision message from the manager. At some point, the agent has to send a PREPARE message to the processes of its database as part of the local 3PC protocol. This message has to be sent either before or after the agent receives the COMMIT message from the global transaction manager.

Suppose the agent sends out the PREPARE message before it has received the COMMIT/ABORT message from the manager. Consider the scenario depicted in Figure 5. There are two participant databases DB1 and DB2, where DB2 uses a local 3PC. DB1 has 3 processes: 2 database processes (db-process11 and db-process12), and one agent (agent 1); similarly for DB2. The global manager sends the initial EXEC and DB1's agent receives it, obtains local agreement to abort and replies NOK to the manager. In the meantime, DB2's agent also receives the manager's EXEC and sends a PREPARE to its local participants. Assume that all sites fail except for the participant sites of DB2 (i.e., the global manager, all agents, and all DB1 sites fail). Since the processes residing in sites of DB2 are in the *prepare* state, after they timeout they contact each other and decide to commit. Thus, this protocol fails.

Consider now what happens if the agent sends the PREPARE message after it receives the global manager's COMMIT message. Again, we have two local databases DB1 and DB2, DB2 being a 3PC one. Now, let us examine the sequence of events shown in Figure 6. The global transaction manager sends the initial EXEC, and the agents for both DB1 and DB2 obtain local agreement to commit and send an OK to the manager. The manager then sends a COMMIT to both agents. The agent at DB1 receives the COMMIT message and makes sure its database commits. However, before DB2's agent can receive the COMMIT message we have the same widespread failure as before. The processes residing in sites of DB2 are all in the *wait* state at this point. DB2 processes can only decide among themselves what to do, which is to abort. Thus, once again the protocol fails.

The reason that a straightforward application of 2PC can not work as global protocol is that the protocol states in the situation just described do not have their usual meanings. Normally, the *wait* state represents the willingness of

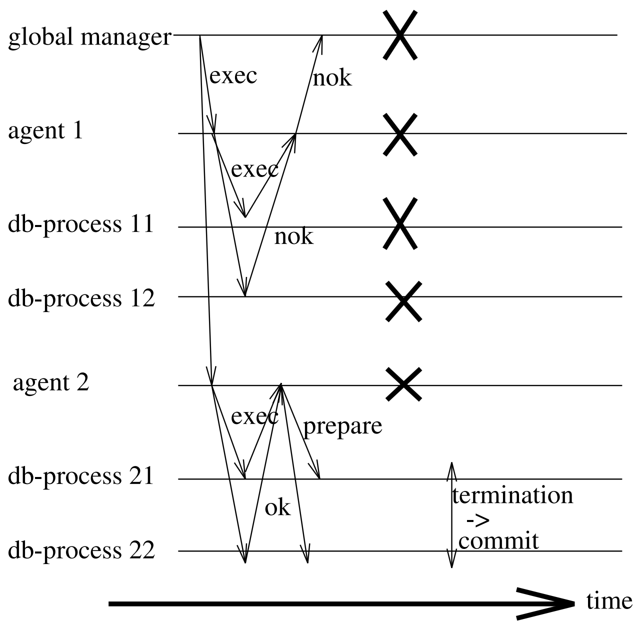

Figure 5: 2PC as the Global Protocol - Case I

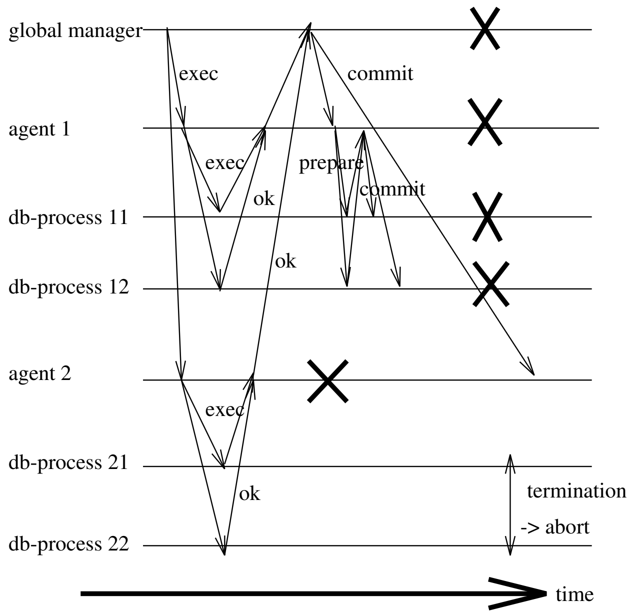

Figure 6: 2PC as the Global Protocol - Case II

the participant to commit while the *prepare* state represents the willingness of the other participants to commit. However, in the first case above, the *prepare* state does not always represent the willingness of participants in other databases to commit, while in the second case, the willingness to commit of the non-local participants is not represented.

Having ruled out 2PC, let us try to use 3PC as a global protocol. Consider the following commit protocol: The global manager sends out the initial EXEC. Each agent sends the EXEC message to the local processes. When the agent receives the OK responses from the local processes, it sends OK to the global transaction manager. After obtaining the PREPARE command from the global manager, the agent relays it to the local processes only if the local database uses a 3PC protocol; otherwise it immediately sends the ACK. When the agent receives the COMMIT message from the global manager, it forwards the message to the local processes.

Assume now that DB1 and DB2, both using 3PC, take part in the transaction, as illustrated in Figure 7. After receiving the EXEC command and polling their local processes, both databases' agents answer OK. All processes are now in a *wait* state. The global manager now sends a PREPARE command. The agent at DB2 receives it, sends out a PREPARE message to the local processes, which then enter the *prepare* state. Meanwhile, the agent at DB1 fails and does not receive the manager's PREPARE message. At this point, DB2's agent also fails. The processes in DB2 run the termination protocol and commit. However, when the processes in DB1 run the termination protocol they abort. Again, the protocol fails.

The problem is that in a 3PC protocol one assumes that each participant knows where the other participants are and can always reach them. However, here a process does not know about processes in other participating databases, and thus cannot reach them during the termination protocol. In effect, by having the local processes communicate information to the global coordinator only through their agent, a single site failure (the agent's) is equivalent to a network partition. So, although our failure model allows only site failures, we reach a situation resembling a communication failure, which 3PC does not tolerate (see [2]).

We must ensure that site failures are not equivalent to network partition. We can start by having every process know about every other participating process. This simple fix does not work. For example, even if we assume that we can send the identities of all the participants as part of the EXEC message, we still run into difficult translation problems: participants belonging to different databases employ different message formats, the same global transaction will result in different transaction ID's locally, the databases possibly use dissimilar election protocols, etc. Moreover, for some databases, the local coordinator

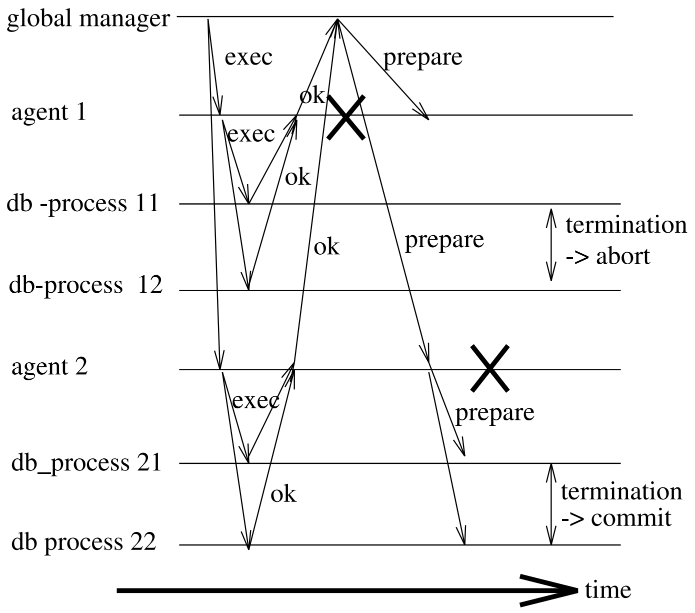

Figure 7: 3PC as the Global Protocol - Case I

might still be alive, and thus their participants are not ready to enter an election protocol.

We now add yet another modification to the protocol just presented. We place an auxiliary process at every site of every database as shown in Figure 8. These auxiliary processes act just like another process in the local distributed database, and participate in the local commit protocol. But these auxiliary processes are actually aware of the existence of the global manager, the agents and other auxiliary processes in the system. Therefore, in time of trouble, they can contact these processes.

In the absence of failures, our new protocol is identical to the 3PC one presented above. Consider now what happens if one of the agents of the 3PC databases fails. At that point, the local participants plus the associated auxiliary processes undertake an election protocol to select the new transaction coordinator. The selected coordinator could be one of the local database processes or one of the auxiliaries. The difficult case is if the selected coordinator is one of the former, all of which are unaware of the global nature of the transaction. When the new coordinator asks all the local database processes and auxiliaries what their state is, the database processes answer truthfully, but the auxiliaries may have to do some work before answering. Each auxiliary process may have to contact its *cohort* (i.e., the global manager, remote agents, and remote auxiliaries) to determine the state of the global transaction.

If an auxiliary process is in the *prepare* state, it informs the local coordinator about it. If the auxiliary process is in the *wait* state, it must find out

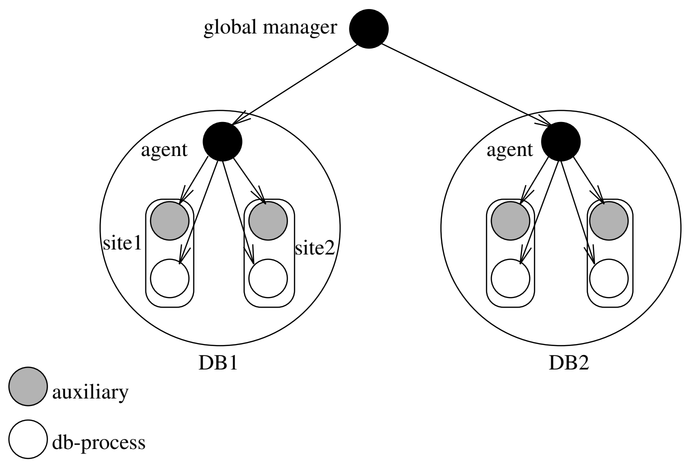

Figure 8: The Modified Model

if there is another process in its cohort which is in a *prepare* state. If so, the auxiliary reacts as though its state is *prepare*. This is done to guarantee that the local database does not decide to abort while other databases decide to commit. By answering *prepare* to the local coordinator the auxiliary ensures that its local database proceeds to commit the global transaction. Alternatively, if the auxiliary cannot find anyone else who is in the *prepare* state, it tells the local coordinator that the auxiliary is in a *wait* state, which results in an abort decision by the local coordinator.

However, there is still a problem (depicted in Figure 9). Suppose that we have two 3PC databases. The global manager reaches the *prepare* state. It sends the PREPARE message to the agents. At this point, both the global manager and the second database's agent fail. The agent of the first database receives the PREPARE, sends it to all the local processes. Now the first agent dies. The first database's processes (real and auxiliary) proceed to commit (since the auxiliaries are in the *prepare* state they agree to this). Now, all of the processes related to the first database die. At this time the participants at the second database realize their agent is gone. The auxiliaries there try to contact their cohort processes, but they cannot find any in a *prepare* state (all of the cohort processes in the first database are dead). The second database proceeds to abort, and now the protocol fails. The problem is that some of the auxiliary processes are not aware that others are in a *prepare* state. Thus, under failure conditions, some remote auxiliaries might have forced a previous commit.

Finally, we present a correct global protocol. As before, we place an auxiliary process at every site of every database, and each auxiliary acts as a regular participant. However, there is one crucial difference with the protocol

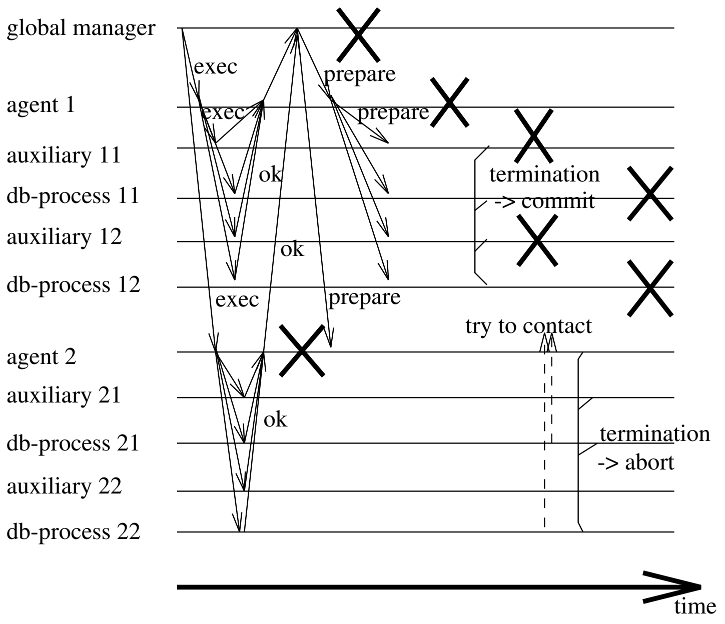

Figure 9: 3PC as the Global Protocol - Case II

presented above. The auxiliary processes are *all* going to transfer to a new state, the *prepare_to_prepare* state, before any of the real database processes moves to the *prepare* state.

We now describe in more detail the actions of our protocol when the local database follows a 3PC local protocol. (We address the 2PC case later.) The protocol is also sketched in Figure 10. During local transaction processing the auxiliary processes do not have work to do. In a global transaction, the global manager sends an EXEC to the agents. The agents send a local EXEC to the database participants and to the local auxiliary processes. If they all reply OK, the agent sends an OK to the global manager. If the latter receives OK from all the agents, it sends a PREPARE_TO_PREPARE message to all the agents, who then send an equivalent message to the auxiliaries (but not to the other participants). The auxiliaries acknowledge the PREPARE_TO_PREPARE, and once their agent receives their acknowledgements, it informs the global manager of that fact. The global manager now sends a PREPARE to all the agents, who forward it to all the local participants (including the auxiliaries). Everyone sends an ACK to the agent, who forwards it to the global manager. When ACKs are received from all agents, the global manager issues a COMMIT message to all agents, who in turn send COMMITs to every local participant.

Consider what happens during a site failure. There are three possible cases: (1) the site may contain the global manager; (2) the site may hold a participating database process and its associated auxiliary; or (3) the site

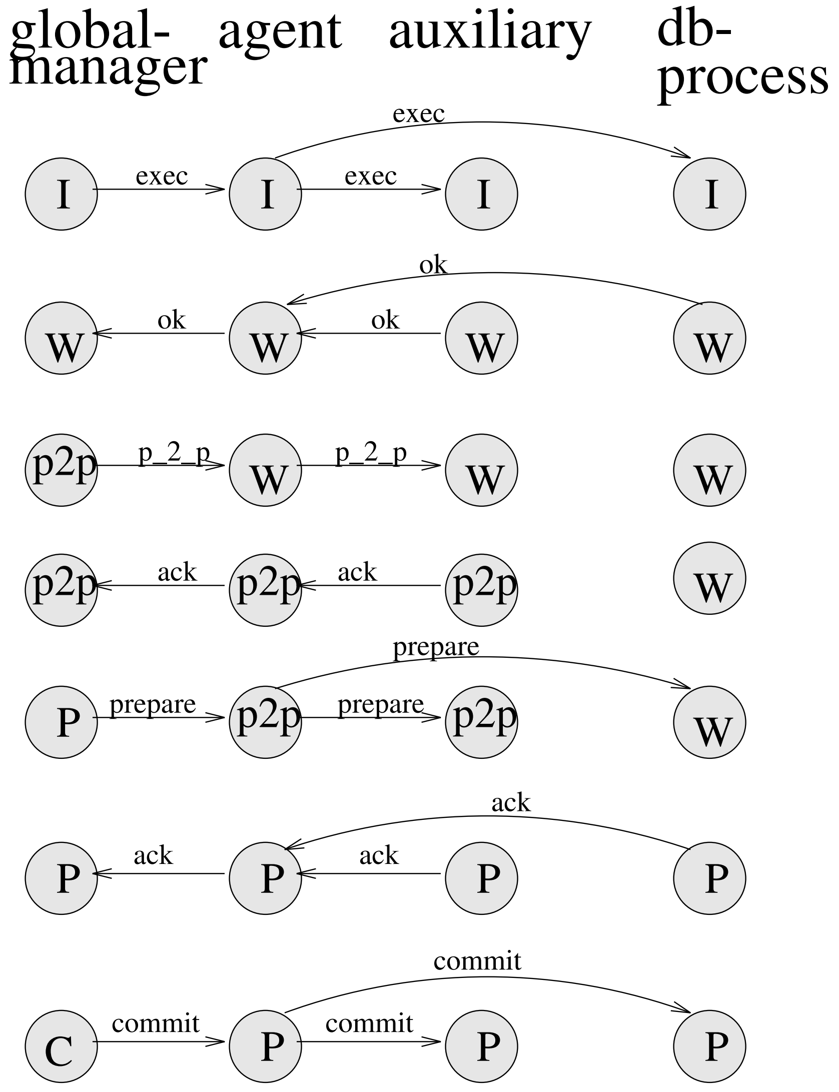

Figure 10: The Correct Global Protocol

may have the agent for that database (and possibly a participating database process). If the global manager's site fails, the agents and auxiliaries follow the usual 3PC election algorithm to select a new global manager. In (2), the auxiliaries follow the local commit protocol. If the failed site held the agent process, the auxiliary processes have extra work to do. In this last case all the processes in the database (as well as all local auxiliaries) enter into the usual 3PC election protocol to select the new local coordinator. The election protocol results in a new coordinator (which may or may not be an auxiliary process). If an auxiliary is selected, it becomes the new agent and all proceeds as before. The difficult case occurs when a regular database process is selected as coordinator. Indeed, we now have a coordinator that has no idea that the ongoing transaction is a global one. So, when it enters the termination protocol it will not contact other databases for consensus.

When a non-auxiliary is chosen as coordinator, the auxiliary process at the new coordinator's site becomes the agent for the database. When the local termination protocol starts, the newly selected coordinator asks everyone what state they are in. The auxiliary process that became the new agent now has an additional task.

If this new agent is in the *wait* state it contacts the global manager and registers itself as the new agent. The next message from the global manager is either a PREPARE_TO_PREPARE or an ABORT. (Remember, a global manager failure at any point results in the election of a new one, so the agent will always eventually receive either one of the two messages). If an ABORT is received, the new agent tells the coordinator that its state was *wait*, which forces a local abort. If a PREPARE_TO_PREPARE is received, the new agent goes to the *prepare_to_prepare* state, and forwards this message to all other auxiliaries at its database. Upon acknowledgement from all other auxiliaries, the agent forwards an acknowledgement to the global manager. When the global manager receives the acknowledgements from all the agents, it then sends a PREPARE back to them. At this point the agent can finally reply to the local coordinator. The agent tells the latter that the agent is in a *prepare* state, which forces a local commit. Note that an agent being in a *prepare* state implies that all others in the cohort are at least in *prepare_to_prepare* state (i.e., either in *prepare_to_prepare*, *prepare*, or *commit*). Therefore none of the other databases can abort this transaction.

However, if the new agent is in the *prepare_to_prepare* state it again contacts the global manager and registers itself as the new agent. The new agent then acts as if it had just received the PREPARE_TO_PREPARE message in the above case.

If the new agent is in the *prepare*, *commit*, or *abort* state, it again registers itself with the global manager. The agent then communicates its state to the local coordinator.

2PC databases are easy to integrate into the above scheme. As we indicated in the previous section, there is also an agent at each 2PC database. If the agent fails, the local processes will block, and processing will stop until the agent process recovers, in which case it will follow the usual 2PC recovery procedure.

Finally, we consider the case of total failure of a database. When a participant at the recovered machine starts up, it looks for another process who can tell it what the result of the transaction was. If an auxiliary process is contacted, the latter will know to ask the global transaction manager what to do. If another participant is contacted (and can give an answer), that participant must have also contacted an auxiliary process. The reason for this is as follows. In 3PC, the only participant that can determine (by itself) the outcome of a transaction is the participant that can be sure it died last. But since every participant and its associated auxiliary process die at the same time (remember, we assume site failures only), the participant cannot be sure it was the last process alive. Only auxiliaries are willing to tell others what the outcome of the transaction was.

For a Pascal-like description of the protocol just presented we refer the reader to [18].

Two final comments. First, since the global manager might fail at any time in the middle of a transaction, multiple election rounds might take place. Clearly, this process can take a long time. Therefore, we have to increase the timeout period of the local databases to a suitably large value. (The timeout period of a database is not usually hard-coded.) In theory, we need an interval large enough to allow n global manager failures, where n is the number of sites in the system. In practice, one would need to worry about only a reasonable number of failures.

Secondly, even if all the participating databases are 2PC, the participants may wish to use the global commit algorithm presented in this section instead of the 2PC one of the previous section for autonomy reasons. The 2PC global algorithm may force a database to block because of the actions of a remote site (say, the global coordinator failing), which has a definite impact on the autonomy of the participating databases. Since the protocol just presented does not block a database upon failure of another one, it may be the algorithm of choice for autonomous environments.

It is not hard to verify that the protocol presented is non-blocking, and we state this below as a theorem. (We consider 3PC databases only.)

Theorem 2 *In the absence of total failures, our algorithm and its termination protocol never cause processes to block.*

Proof: If there are no failures, it is trivial to show that the protocol is non-blocking. We examine now the situation where failures occur. First, consider the case where the global manager has not failed. If there are local database failures, the local (non-blocking) termination protocol drops these processes from consideration and the protocol continues. If there is a failure at the site of the local coordinator, the local processes elect a new local coordinator and proceed. The auxiliary at the same site becomes the new agent, and the global manager can continue its protocol communicating with it, until a decision is reached.

We now turn to the situation where the global manager fails. The agents and auxiliaries initiate the global termination protocol and elect a new global manager. The elected global manager asks the agents for their local states. Each possible combination of states (or lack of response from a failed agent or auxiliary) causes the manager to activate a single termination rule. Thus, if the newly elected manager does not fail, the operating processes reach a decision and there is no blocking. Otherwise, a new invocation of the termination protocol will be initiated until all processes reach a decision or there is a total failure. □

We now prove the correctness of the integrated commit protocol described above. Recall the five requirements for correctness listed in Figure 2. We first discuss Requirements 2-5 in Theorem 3, and then address Requirement 1.

Theorem 3 *The proposed protocol satisfies the correctness Requirements 2-5 given in Figure 2.*

Proof: Requirement 2 states that once a process makes a decision it may not reverse it. Assume a process makes its decision. If the process does not fail, by inspection of the protocol it is easy to see that the process can never undo the commit/abort decision. If the process fails after making the decision, it will read its log and decide in the same manner.

We now verify the third correctness criterion (i.e., the commit decision can only be reached if all processes voted OK). Suppose that a process voted NOK (or did not vote). If that fact is noted by the local agent, it will communicate it to the global manager and an abort decision will be reached. If the agent fails before contacting the global manager, the latter will decide to abort. If the global manager failed before receiving word of the NOK vote, it would never have told any agents to move to the *prepare_to_prepare* state. Hence, when a new global manager is chosen, it will decide to abort the transaction.

Requirement 4 says that, if there are no failures and all processes voted OK, a commit decision will be reached. It is easy to verify this requirement by inspection of the protocol.

The last criterion requires us to show that, given any execution schedule containing failures, if all failures are repaired and no new failures occur for a sufficiently long time, then all processes will eventually reach a decision. If a process has not failed, then by Theorem 2 (which states the protocol is non-blocking) it will eventually reach a decision. If the process failed, when it recovers it reads in its log its state at the time of failure. If it failed before it sent OK, or if it sent NOK, it decides to abort. If it received COMMIT or ABORT before it failed, it can also make a decision. Otherwise, the process needs the help of other processes. Recall that our protocol tolerates site failures only. Therefore, when a site recovers, both the database process and the auxiliary process at that site are recovered. The auxiliary process can communicate with the processes in its cohort and ask for the decision, even if all sites in the local database failed. If there is no total failure, then (by Theorem 2) a decision either has already been made or is being made and the auxiliary will eventually learn about it and adopt the decision. Once the auxiliary knows the decision, it will relay it to the database process. In a total failure situation, blocking takes place until: a process can recover independently, a process knows it was the last process to fail, or all processes are repaired. In the first case, the process eventually passes the decision along to others. In the second case, the process knows which was the final global manager and waits for it to recover and then learns of its decision (either the global manager already reached a decision or it will do so after executing the termination protocol). In the last case, the processes wait for all others to recover. Then the final global manager can be identified, and it can inform others of its decision or execute the termination protocol. □

We now consider Requirement 1, i.e., all processes that reach a decision reach the same one. Proving Requirement 1 for 2PC databases poses no difficulty. Clearly, there will not be a disagreement between two processes in the 2PC database by the correctness of 2PC. We claim that no 2PC process can disagree with the global manager, since the 2PC database agrees to commit or abort based only on the global manager's instructions; remember that, unlike 3PC databases, 2PC databases block upon local coordinator (the agent) failure, and wait until it is repaired to continue. Below we prove that the global manager will always agree with 3PC processes. Thus, there is no possibility of a disagreement between a 2PC and a 3PC process. To simplify the proofs below, we now omit mention of 2PC databases in them.

We divide time into *intervals* between the elections of global managers. We proceed in two steps. The first step (*Theorem 5*) proves that all decisions taken within a specific interval are consistent. The second step (*Theorem 8*) proves that decisions taken at any interval are consistent with those taken in earlier intervals. We consider a local coordinator as having committed at the i^{th} interval if its agent has consulted with the i^{th} global manager before reporting to the local coordinator (and this report enables the coordinator to make a decision). Note that the actual decision can be made after the failure of that specific global manager.

Lemma 4 *When a local coordinator decides to commit, its agent reported either a* prepare *or a* commit *state. When a local coordinator decides to abort, its agent reported either an* abort *or a* wait *state.*

Proof: By induction on the number of failures of the local coordinator in a specific database. (We refer to the period between two consecutive local coordinator failures as a *phase*.)

Phase 0: Before the first failure has occurred, the local coordinator is the agent. Thus, it will decide to commit only if its state is *commit*, and likewise for an abort.

Phase i: Suppose first that the local coordinator decides to commit. At least one process reported a *prepare* state or a *commit* state. That process must have received a PREPARE message in an earlier phase. Look at the first phase in which any process in the database got a PREPARE message. According to our algorithm, at that time all auxiliaries had to be in at least a *prepare_to_prepare* state. Therefore, at the i^{th} phase the agent, which is an auxiliary, cannot report a *wait* state or an *abort* state.

Now suppose that the local coordinator decides to abort. Then either some process reported an *abort* state or all processes reported a *wait* state. In the latter case we are done. However, if some process reported *abort* then there are two possibilities. The first one is that the process voted to abort (NOK). In this case, it is not hard to check that no auxiliary could move from a *wait* state to a *prepare* or a *prepare_to_prepare* state. This is so because either the NOK vote was sent to the global manager by an agent (while this agent was acting as local coordinator), or the global manager never received an OK from the original agent. In either case the global manager will abort the transaction. The second possibility is that an ABORT message was received by the reporting process in an earlier phase. By the induction hypothesis, the agent at this phase is in either in an *abort* state or in a *wait* state. Since it is not in a *prepare_to_prepare* state, no other auxiliary process can be in a *prepare_to_prepare* state (because the agent is the one which gives the instruction to move to the latter state). But the reason the agent is in an *abort* or a *wait* state is because the global manager told it to reply in such a manner. Therefore, the global manager will not tell any auxiliaries to enter the *prepare_to_prepare* state. Thus, all auxiliaries (including the i^{th} agent) will not proceed to the *prepare_to_prepare* state. □

Thus, we have shown that the agent can determine the outcome of a local transaction. We now show that, between global manager failures, all processes agree.

Theorem 5 *All processes that reach a decision within the i^{th} interval, reach the same one.*

Proof:
At the i^{th} interval all agents consult with the same global manager which will not report contradictory decisions. By Lemma 4 we know that the agent can determine the local coordinator's decision. Since agents do not obtain contradicting information and since the local decisions are based on the agents' information, all local databases that reach a decision, reach the same one. $\square$

Lemma 6 *If any operational auxiliary process is in a wait state then no other process (whether operational or failed) can have decided to commit.*

Proof: It is obviously true before any termination protocol starts (local or global). We will verify that, if the above holds before a termination protocol starts, it will hold after even a partial execution of that protocol. Suppose by the way of contradiction that a decision to commit was taken before. If the global manager was the one to take the decision, then according to our protocol, all operational auxiliary processes must have moved first to a *prepare_to_prepare* state. If a local manager was the one to make the decision, then its associated agent must have reported at least a *prepare* state. But again, an agent can do so only after all operational auxiliary processes have moved to a *prepare_to_prepare* state first. $\square$

Lemma 7 *Consider the i^{th} interval ($i > 0$). If a process p that is operational during at least part of this time is in* prepare_to_prepare *state, then some process q that was operational in $(i-1)^{th}$ interval was in* prepare_to_prepare *state then.*

Proof: At the beginning of the i^{th} interval, when the new global manager is selected, if it did not find at least one process in *prepare_to_prepare* state the global manager would decide to abort the transaction at that point, and thus would not send PREPARE_TO_PREPARE messages to any process. $\square$

Theorem 8 *Under our protocol, all operational processes reach the same decision.*

Proof: by induction on i, the i^{th} interval.
For $i = 0$ (before the first global manager's failure): We know from Theorem 5 that all processes that reach a decision in an interval reach the same conclusion.

For $i > 0$: By Theorem 5, all those processes which reach a decision within the i^{th} interval, reach the same one. It remains to show that this decision is consistent with previous ones. We distinguish between the different termination rules that are applied when making the decision. Recall that a decision might be made either by the global manager or by a local coordinator during this interval.

Suppose first that a decision is made to abort. Assume also that the reason to abort is that some relevant process reported its state as being *abort*. If some process reports this state because it has voted NOK or it has not voted, then it is not hard to see that a decision to commit could not have been made earlier (this follows from Requirement 3 which we already proved). The other possibility is that a process reports *abort* because it received an ABORT message earlier. If it received the ABORT message before the current failure of the global protocol then, by the induction hypothesis, since this process decided to abort no process could have decided in earlier invocations to commit. However, if the ABORT message was received during this interval, suppose by way of contradiction that some process decided to commit in an earlier interval. In that interval all auxiliaries had to be in at least *prepare_to_prepare* state, and therefore no global manager would decide to abort or any agent force its local coordinator to abort (note that we showed in Lemma 4 that the agent forces its local coordinator's decision). Therefore, none of the managers/coordinators could have sent an ABORT message since then.

Assume now that a decision to abort was made because all relevant processes have reported a *wait* state. By Lemma 6, no process can have previously committed.

Consider now the case where a decision is made to commit. If a decision to commit is taken because some operational process reports *commit*, it is again not difficult to see that an abort decision could not have been made earlier. The other possibility is that a decision to commit is made because some process reports *prepare_to_prepare* or *prepare* to the global manager or *prepare* to the local coordinator. In the former case the global manager will make sure before making a decision that all the auxiliary processes go to a *prepare_to_prepare* state. In the latter case, we know that the auxiliaries have already gone through a *prepare_to_prepare* state. Now suppose that at an earlier interval some process decided to abort. But in that interval there was at least one process in *prepare_to_prepare* state by Lemma 7. Thus, the global manager could not have decided to abort, nor would it have instructed any agent to force a local abort. □

6 Summary

We have discussed the problem of atomic commitment in a system of heterogeneous distributed databases. We examined two cases. The first was that of non-externalized commit databases, for which we only presented a brief literature survey. The second case, which we have explored in detail, was that of externalized commit databases.

We presented a novel global commit protocol for heterogeneous databases composed of distributed databases using any of various 2PC protocols. We did this by having our global coordinator engage in a centralized 2PC with agent processes at the participating databases, and showing that we could integrate the local 2PC protocols into our global one. The advantages of our global protocol are its simplicity and that it addresses the most important case in practice.

We then considered database systems using either 3PC or 2PC protocols. In developing the global commit protocol for this case we required the use of helping processes at each local database. The overhead of such processes, as well as the additional exchanged messages and the increased time-out intervals that our algorithm requires in a real implementation are the negative aspects of our solution. The main contribution of our work is that we have shown how to integrate a heterogeneous collection of databases using most of the commonly studied commit protocols. Furthermore, our protocol does not block a database because of remote failures, thus preserving local autonomy. We expect autonomy to become especially important in very large heterogeneous database environments.

Our future research plans in this area involve the implementation of a prototype heterogeneous database for a mobile computing environment. Our prototype will integrate a variety of commercially available databases, as well as some non-database information sources (e.g., file systems, mail servers) being developed at MITL. We expect that some of the non-traditional data sources we are using will employ a variety of non-blocking protocols. (In a mobile environment there is a need for support of autonomous operation in the face of potentially frequent failures, which forces the use of non-blocking protocols.) We expect that the work described in this paper will be of help in integrating those non-blocking protocols.

Acknowledgements

We would like to thank the following people for their advice and review of our paper: Matt Blaze, Daniel Barbara, Hank Korth, Hector Garcia-Molina, and the anonymous referees.

References

[1] K. Barker and M.Tamer Özsu. "Reliable Transaction Execution in Multidatabase Systems," *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pp. 344-347, 1991.

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

[3] Y. Breitbart, H. Garcia-Molina and A. Silberschatz. "Overview of Multidatabase Transaction Management," *Technical Report STAN-CS-92-1432, Department of Computer Science, Stanford University*, 1992.

[4] Y. Breitbart, A. Silberchatz and G. Thompson. "Reliable Transaction Management in a Multidatabase System," *Proceedings ACM SIGMOD*, 1990.

[5] H. Garcia-Molina, K. Salem. "SAGAS," *Proceedings ACM SIGMOD*, pp. 249- 259, 1987.

[6] D. Georgakopoulos. "Multidatabase Recoverability and Recovery," *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pp. 348-355, 1991.

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

[8] A. Gupta. Integration of Information Systems: Bridging Heterogeneous Databases. *IEEE Press* 1989.

[9] Formal and Protocol Reference Manual: Architecture Logic for LU Type 6.2. *IBM manual SC30-3269-3* December 1985.

[10] B. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. *Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center*, 1976.

[11] J.G. Mullen, A.K. Elmagarmid, W. Kim and J. Sharif-Askary "On the Impossibility of Atomic Commitment in Multidatabase Systems," *Proceedings of the 2nd Intl. Conference on Systems Integration*, pp. 625-634, 1992.

[12] P. Muth and T.C. Rakow. "Atomic Commitment for Integrated Database Systems," *Proceedings of the 7th Intl. Conference on Data Engineering*, pp. 296-304, 1991.

[13] D.J. Rosenkrantz, R.E. Stearns and P.M. Lewis. II System Level Concurrency Control for Distributed Database Systems. *ACM Trans. on Database Systems* 3(2) p. 178-198, June 1978.

[14] D. Skeen. Nonblocking Commit Protocols. *Proc. ACM SIGMOD Conf. on Management of Data*, p. 133-147. June, 1982.

[15] D. Skeen. A Quorum Based Commit Protocol. *Proc. 6th Berkeley Workshop on Distributed data Management and Computer Networks, ACM/IEEE*, p. 69-80, February, 1982.

[16] D. Skeen. Crash Recovery in a Distributed Database System. *Technical Report, Memorandum No. UCB/ERL M82/45, Electronics Research Laboratory, University of California at Berkeley*, 1982.

[17] Sybase, Inc. Open Client DB-Library/C Reference Manual, Release 4.6: Revised September, 1991.

[18] A. Tal and R. Alonso. "Integrating Commit Protocols in Heterogeneous Database Systems," *Technical Report CS-TR-375-92, Department of Computer Science, Princeton University*, June, 1992.

[19] F. Upton IV. OSI Distributed Transaction Processing, An Overview. *Workshop on High Performance Transaction Processing* September, 1991, Asilomar.

[20] A. Wolski and J. Veijalainen. "2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase," *Proceedings of PARBASE-90*, 1990.

A Commit Protocols

In this appendix we present simplified descriptions of the commit protocols which we consider in the paper. We concentrate on several variants of the 2PC protocol and on the 3PC protocol. For more details on this subject see [2].

A.1 Centralized 2PC

Centralized 2PC is described in [7] and [10]. One of the participants acts as coordinator. The algorithm is as follows:

1. The coordinator sends an EXEC message to all participants.

2. When a participant receives an EXEC, it responds by sending the coordinator the participant's vote: either OK or NOK.

3. The coordinator decides to commit and sends the COMMIT message to all participants if all votes are OK and the coordinator does not timeout. Otherwise, the coordinator sends an ABORT message.

4. Each participant waits for the COMMIT or ABORT message and acts accordingly.

The processes which belong to a 2PC database can be in one of four states: an *initial* state (before they get the EXEC command), a *wait* state (when they respond with OK), a *commit* state (when they receive the COMMIT message), and an *abort* state.

A.2 Decentralized 2PC

The decentralized 2PC protocol is presented in [14]. All participating processes must communicate (i.e., the communication topology is a complete graph). The coordinator's role now is only to start the algorithm, not make the commit decision by itself. The algorithm is:

1. The coordinator sends its vote to all participants.

2. Each participant responds by sending its own vote to all other processes.

3. After receiving all the votes, each process makes a decision.

The states the processes can be in are the same as in the centralized version.

A.3 Linear 2PC

The linear 2PC protocol appears in [7] and [13]. Processes are linearly ordered, and each needs only to communicate with its neighbors. The algorithm is as follows:

1. Each process waits for the vote from its left neighbor (except for the leftmost process which starts the algorithm). If it receives an OK and its own vote is OK, it forwards the OK vote to its right neighbor.

2. The rightmost process then makes the decision. It sends a COMMIT or an ABORT to its left neighbor. Each process forwards the message to its left neighbor.

Once again, the states the processes can be in are the same as in the centralized 2PC case.

A.4 Hierarchical 2PC

The participants are ordered in a tree. A participant communicates only with its parents and its children. The root acts as the coordinator. The algorithm is as follows:

1. The coordinator sends an EXEC message to all its children.

2. When a participant receives an EXEC, it forwards it to its children. A leaf participant responds by sending the parent the participant's vote: either OK or NOK.

3. If all the votes received by a participant from its children are OK and the participant itself votes OK, the participant forwards OK to its parent. Otherwise, it sends NOK.

4. The coordinator decides to commit and sends the COMMIT message to all its children if all votes are OK and the coordinator does not timeout. Otherwise, the coordinator sends an ABORT message.

5. Each participant waits for the COMMIT or ABORT message from its parent. When the participant receives the message, the participant forwards it to its children and acts accordingly (committing or aborting the transaction).

Again, the protocol states are the same as above.

A.5 2PC Recovery Protocol

1. If the process recovering was the coordinator and if the log contains a COMMIT or an ABORT record, then the coordinator had decided before the failure. Otherwise, the coordinator decides to abort by writing an ABORT record in the log.

2. If the process recovering was not the coordinator for the transaction then:

   - If the log contains a COMMIT or an ABORT record, then the participant had reached its decision before the failure.
   - If the log does not contain an OK record, the process decides to abort.
   - Otherwise, the process tries to reach a decision using the termination protocol (i.e., the process blocks until it can contact another process that is aware of the decision).

A.6 3PC Protocol

The 3PC protocol is described in [14], [15] and [16]. It has the advantage of dealing with coordinator failure without blocking. The algorithm is given below:

1. The coordinator sends an EXEC message to all participants.
2. When a participant receives an EXEC, it responds by sending the coordinator the participant's vote: either OK or NOK.
3. The coordinator decides to commit and sends the PREPARE message to all participants if all votes are OK and the coordinator does not timeout. Otherwise, the coordinator sends an ABORT message.
4. When a participant receives a PREPARE message, it responds by sending the coordinator an ACK.
5. Each participant waits for the COMMIT or ABORT message from the coordinator and acts accordingly.

In case of coordinator failure, a new coordinator is chosen (i.e., an *election* protocol is followed), and then the algorithm enters its *termination* protocol:

1. If some process is aborted, the coordinator decides on abort.
2. If some process is committed, the coordinator decides on commit.
3. If no process has received the PREPARE message, the coordinator decides to abort.
4. If some process has received the PREPARE message, the coordinator continues the protocol.

The processes can be in any one of the states mentioned in the 2PC case or in a new state called the *prepare* state (when the process acknowledges the PREPARE message).

A.7 3PC Recovery Protocol

1. If the recovering participant determines that it had failed before sending OK, it decides to ABORT.
2. If the recovering participant sees that it had failed after receiving a COMMIT or ABORT, it knows that it had already decided.
3. Otherwise, the participant must ask other processes about their decision.

The coordinator's actions are similar.