Integration of Commit Protocols in Heterogeneous Databases *

Ayellet Tal
Princeton University

Rafael Alonso
Matsushita Information Technology Laboratory

September 16, 1992

Abstract

In this paper we explore the problem of atomic commitment among a collection of heterogeneous distributed databases. We distinguish between two cases. In the first, the individual databases which participate in the system do not externalize their commit protocols. Several ways to handle this case have been proposed in the literature. We discuss and classify the various solutions. The second case, where the individual databases do externalize their commit protocols, has not been discussed in detail in the past. We will focus on this case, and propose new heterogeneous commit protocols for two subproblems: joining databases that have different 2-phase commit protocols (e.g., centralized and linear); and uniting databases using a 3-phase commit protocol with others employing a 2-phase commit protocol.

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

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 already exist and need to cooperate with

*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, 182 Nassau Street, 3rd Floor, Princeton, NJ 08542, (609) 497-4600, alonso@mitl.com

each other. These environments, and the problems involved in integrating a heterogeneous collection of databases have already been discussed in the literature (for example, see [7]). There are many open problems. Among them are semantic incompleteness and inconsistency, naming services, security, concurrency control, deadlock detection and atomic commitment.

In this paper we address the problem of atomic commitment for autonomous heterogeneous databases. This area has not been fully explored in the past. For example, the work on federated databases [8] and multidatabases [11] does not address the topic of commit protocols in full detail. The work on super-databases [12] does concern itself with commit protocols, but that research is less appropriate for an environment of autonomous database management systems, since it imposes a large degree of cooperation among the local systems; for example, the databases need to make public the exact serial ordering of their local transactions.

If we consider the integration of autonomous databases, we must distinguish between two kinds of systems. The first consists of databases which externalize their commit protocols (we refer to them as *externalized commit databases*). These systems make public whether they use 2-phase commit (*2PC*), 3-phase commit (*3PC*), a version of the Byzantine agreement protocol, etc. They also make available the internal state of the commit protocols (i.e., wait state, prepare state, etc.) but they need not disclose their internal state pertaining to other database activities (such as the commit ordering of transactions). The second kind of databases are those which do not externalize their commit protocols (*non-externalized commit databases*). Most previous work in the heterogeneous database literature has addressed the case of non-externalized commit databases; we will briefly survey this topic in a later section. However, the main topic of this paper is atomic commitment for databases which do externalize the commit protocols.

We believe the importance of these systems will increase because they will become more prevalent. This is so because we expect that, in the near future, centralized databases will be replaced by distributed ones; distributed databases usually externalize their commit protocols. Unfortunately, externalized commit databases have not been examined in detail in the past. The work in this area has usually consisted of the various standardization efforts supported by industry (i.e., LU 6.2 [9], or OSI's TP service [17]). 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.

In this paper we will describe the integration problem for externalized commit databases, show why it is not trivial to solve via some examples, and suggest novel protocols for heterogeneous commit. We present two new protocols. The first is to be used in an environment where all the databases have some type of 2PC commit protocol (i.e., centralized, linear, etc.). The more difficult case of integrating 3PC and 2PC databases is also studied. The main advantage 3PC has over 2PC is that it is a nonblocking protocol. While 3PC has not been used extensively in the past, it (or other similar non-blocking protocols) might become more common in heterogeneous environments where site autonomy is an important requirement. A new commit protocol for these environments is presented, one of whose most important features is that a participating database will not block because of a failure at another database. Thus, autonomy is preserved.

In the following section we present a partial overview of previous research in heterogeneous commit protocols for non-externalized databases. In Section 3 we examine externalized commit databases and describe how to merge databases which use a variety of 2PC algorithms. In Section 4, we extend this result to include the merging of 2PC and 3PC systems. We conclude by highlighting the results of this paper and discussing directions for future research. In Appendix A and B we formalize some of the algorithms presented in sections 3 and 4. For the sake of completeness and consistency of terminology, in Appendix C 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. Merging such databases essentially requires combining the local schedules into a global one. In assembling this global schedule one may assume that the local schedules are *locally serializable*, and the goal is to devise an overall schedule that is *globally serializable*. One needs to deal with a number of problems when attempting to construct a global schedule. In this section we describe some of these problems and outline various proposed solutions. 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.

Two new types of scheduling *conflicts* must be handled in integrating non-externalized commit databases. The first are *indirect global conflicts* which are discussed in [2] and [5]. An example of this type of conflict would occur between two seemingly unrelated global transactions which are affected by the actions of local transactions. For instance, consider global transactions G1 and G2, where G1 reads item D at site S2 and writes item A at site S1, and G2 reads item B at site S1 and writes item C at site S2. Suppose there is a local transaction L1 at site S1 which reads A and writes B, and a second local transaction at site S2 which reads C and writes D. If G1 and G2 are not scheduled in the same order at both sites, the resulting global schedule will not be serializable. Further explanations and appropriate solutions to this problem are presented in the references above.

The second type of conflicts are *local-global conflicts* whose existence makes difficult the goal of achieving serializability while not changing the workings of the participating databases. For example, consider object A at site S1 and object B at site S2. Suppose there is a global transaction G1 which reads A, reads B and then writes A and B. Suppose also that there exist two local transactions: L1 at site S1 and L2 at site S2. L1 reads and writes A while L2 reads and writes B. When a decision to commit G1 is taken, one of the sites (either S1 or S2) has to decide to commit first. Suppose S1 is the one. Now S2 fails before it can commit. When S2 recovers, G1 will be aborted at S2 (but G1 at S1 has been committed already). Then L2 is activated and committed, and only then is G1 restarted in S2. Thus, we do not have a serializable schedule. If S2 were to commit first, we would have an equivalent problem.

Various solutions to the local-global conflict problem have been proposed, each of them presenting different tradeoffs. The obvious way to deal with this 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.

A second alternative is to give up the demand for serializability. Instead, one could adopt a different correctness criteria. For example, see [4] where Sagas were introduced, or [3] where quasi-serializability is defined. However, there are many applications where serializability cannot be sacrificed.

A third approach would be to limit the type of transactions allowed in the system. For instance, global read-only transactions are easy to handle, while global one-update transactions can be implemented in a way such that serializability is guaranteed. The latter can be handled by sending the COMMIT message first to the updatable database and, when this database acknowledges, sending the COMMIT message to the other databases. As before, these specialized solutions are not always applicable.

Finally, one may attempt to limit access to the data. Such a solution was suggested by [2]. In their approach the data items are divided into globally-updatable items and locally-updatable items, and transactions modifying globally-updatable data items would not be allowed to read locally-updatable items. The drawback of this approach is the restrictive access to the objects in the databases.

3 Merging 2PC Databases

As we pointed out in Section 2, non-externalized commit databases may be integrated, but at the cost of sacrificing functionality. We now consider the integration of databases that do externalize their commit protocols. Very little work in this area has appeared in the literature. We first discuss the database model we will use in the rest of the paper.

The Model: We assume our heterogeneous database will be built on top of existing distributed databases. A basic assumption of our model is that we are not allowed to make any modifications to the source code of the databases; we have to treat them as black boxes, except that we know what their commit protocols are. We will define a *local* transaction as a transaction which resides in a single distributed database. (Note that a local transaction may be executed in several sites. However, all those sites belong to a single logical database.) A *global* transaction involves several distributed databases. Every global transaction is sent to the *global transaction manager*. The global transaction manager has a server (i.e., the *agent*) at a single site at each distributed database involved in the transaction. The agent at each database is responsible for coordinating the local work on behalf of the global transaction. The individual databases themselves are not aware of whether a given transaction is global or not; only the agent has this knowledge.

When we discuss global protocols in this section (and the next) we assume that the participating distributed databases use one of the commit protocols described in Appendix C as their local commit protocol. Moreover, when we discuss the protocol for 3PC databases in the next section, we will assume a reliable network (because the 3PC protocol is designed to tolerate only site failures). When we discuss 2PC databases only, we will consider communication failures as well.

We will first consider a heterogeneous database composed of only 2PC databases. In practice, this is the most common case. (As a matter of fact, we are not aware of any commercial 3PC database systems.) It is an easier case to consider, and there is less overhead involved in our solution to this case than in the protocol we propose for 3PC databases.

We now show a global protocol to integrate 2PC protocols of multiple databases. The protocol is as follows. The global transaction manager will run a centralized 2PC protocol with the agents as participants. The global manager will play the role of coordinator. Below, we consider each of the possible versions of 2PC and describe how that local algorithm can be meshed with the global one.

- *hierarchical:* If the local database uses a hierarchical 2PC, it is trivial to integrate the local protocol with the global one. Adding a global transaction manager is similar to adding a new root to which all the local roots are connected. The global protocol is obvious. The agent (i.e., the process at each local database acting on behalf of the global transaction manager) waits for a message from the global transaction manager. When the agent gets the message, it forwards the message to the local sites. After collecting the answers from the local sites, the agent returns the answer to the global manager.

- *centralized:* It is also fairly straightforward to integrate a centralized 2PC system. The agent for each database serves as the coordinator for the local transaction. When the global transaction manager sends its request for COMMIT or ABORT, the agent sends the same request to the local database sites (remember the local database is a distributed one). If all the local sites 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 will pass along the global message to the local sites. The one non-trivial detail is that, when the agent site recovers after a failure and finds out that it is in a *wait* state, it should realize that it is not the real coordinator but that instead it should ask the global manager what action (ABORT or COMMIT) to take.

- *decentralized:* The main difficulty in dealing with a decentralized 2PC database is that, in such systems, the initial message that the coordinator sends to the other participants has two roles: it starts the commit process, and it also communicates to the others that the initiating site 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 sites (OK and NOK are the only two kinds of messages it can send), the local sites 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 (say, because another agent at a different database decided to abort). We have a number of ways of dealing with this problem, but we will only mention one here. A possible solution 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 sites (including the auxiliary one). All the local sites will exchange OK or NOK and the agent will then know whether the local database will commit or not. The agent will then send 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.

- *linear:* In linear 2PC, the COMMIT decision is made by the rightmost participant in the linear chain (of course, intermediate participants can force an ABORT). To implement a global centralized 2PC we again need the help of an auxiliary process, which we will force to be the rightmost participant in any commit that involves a global transaction. In other words, our agent process will start the commit process (i.e., act as the leftmost participant); if an intermediate participant decides to abort, the auxiliary process will eventually learn of it and tell the agent, who will then communicate with the global transaction manager. In the event that all participants decide to commit, the auxiliary will tell the agent of this, and the agent will communicate with the global transaction manager, and wait for the latter's reply, either a COMMIT or an ABORT; the agent will then force 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 (say, based on the data items referenced in the transaction), but rather, that the site that starts the commit protocol specifies the linear order in its initiating OK command.

In Appendix A we formalize some of the algorithms presented above using a Pascal-like language similar to that in [1].

We now present a correctness proof for the algorithm just described.

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

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 will first show that our modifications do not affect the correctness of the local protocols. It is easy to see that our claim is true when the local 2PC is either hierarchical or centralized (because we did not modify the local portion of the algorithm other than to add a new site, i.e., the agent). 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.

Consider decentralized 2PC first. As far as the local commit protocol goes, our auxiliary and agent processes act as just two more sites involved in the commit decision. The communication that goes on between the agent and the auxiliary is transparent to the participants, and thus cannot affect their decisions. And of course, the agent and the auxiliary obey all the rules of the protocol (the only difference between the auxiliary and a participant is that the former may take longer in responding OK or NOK, but it will eventually respond).

In the case of linear 2PC, once again the communication between auxiliary and agent is invisible to the other participants (other than perhaps observing a longer than usual delay in responding by the auxiliary process). Since both agent and auxiliary appear as simply two other participant sites, once again our modifications have not affected the correctness of the local algorithm.

All that remains now is to justify the correctness of the global protocol.

¿From the point of view 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. □

A few final comments. In order for the various databases to cooperate with each other, n protocols are needed for n variations of the 2PC protocol. The careful reader may wonder why n^2 protocols are not needed. The answer is that, in our model, we have a single global manager that coordinates the various databases through agents in every database. As a result, each agent needs a protocol to cooperate with the manager, and is unaware of the different commit protocols used by other databases. Therefore, every database has to understand only the single global protocol that it and the global manager will follow. The agent must also worry about generating the appropriate local messages to implement the instructions from the global manager. As for recovery, the agents need only speak to the global manager (and possibly to each other for performance reasons). Clearly, the agents may use a single protocol among them, regardless of the the local protocols of the individual databases. Lastly, one need not use centralized 2PC as the global protocol. Any other 2PC version would do as well.

4 Merging 2PC and 3PC Databases

In this section we consider the more general case of integrating databases that may use 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 transaction manager will engage in a multi-phase protocol with the agents, and the latter will work to integrate the local commit protocols into the global one.

Although we do not formally prove that it is impossible to use a global 2PC protocol between the global manager and the agents, we start this section by presenting the reasoning to suggest why this might be so. We then consider the use of a 3PC protocol as the global one. We will 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 some light on certain features of our protocol. Finally, we present a new global protocol that does perform correctly; this protocol requires an extra phase and the use of auxiliary processes. The impatient reader can skip ahead to the correct protocol by going directly to page 12.

To see why it is not appropriate to use a 2PC algorithm between the global manager and the agents as the basis for the global commit protocol, consider what would happen if we used it. As before, there is an agent process at each database (which again can consist of 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 sites 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 sites of its database as part of the local 3PC protocol. This PREPARE 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 a system with two participant databases DB1 and DB2, where DB2 uses a local 3PC. The global manager sends out 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 (either before or after it replies OK/NOK) sends the PREPARE statement to its local participants. Assume that now there is a widespread failure in the system and all sites fail except for the participant sites of DB2 (i.e., the global manager, all agents, and all DB1 sites fail). Since the sites of DB2 are in the *prepare* state, once they timeout after not receiving any more messages from the agent, they will contact each other and decide to commit. Thus we have an invalid result from our protocol (since DB1 sites decided to abort).

Let us consider now what happens if the agent sends the PREPARE message after it receives the global manager's COMMIT message. As before, we have two local databases DB1 and DB2, DB2 being a 3PC one. Consider the following sequence of events. The global transaction manager sends out 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 (i.e., the global manager, all agents, and all DB1 sites fail). The sites at DB2 are all in the *wait* state at this point (since the agent must have acquired their OK before sending his OK to the global manager which clearly is needed before the latter can send out a COMMIT message). Since all other sites have failed, the DB2 sites can only decide among themselves what to do, and (since they are in the *wait* state) they will decide to abort. Thus, once again we have an incorrect protocol.

As we have shown above, a straightforward application of 2PC will not work as global protocol. The difficulty lies in that the protocol states in the situation just described do not have their usual meanings. Usually, the *wait* state represents the willingness of the site to commit while the *prepare* state represents the willingness of the other sites to commit. However, in the first case above, the *prepare* state does not always represent the willingness of sites in other databases to commit. And in the second case, although the *wait* state represents the willingness of the local site to commit, the willingness to commit of the other sites is not represented.

Since 2PC is not adequate, it is reasonable to consider using 3PC as a global protocol. Our first attempt will consist of a straightforward use of 3PC as global protocol for a collection of 2PC and 3PC databases (in effect, a type of hierarchical version of 3PC). However, this simple solution will not work without additional changes.

Consider the following commit protocol: The global manager sends out the initial EXEC. When each agent gets the global EXEC message, it sends the EXEC message to the local sites participating in the transaction. When the agent receives the OK response from the local sites, it sends OK to the global transaction manager. After obtaining the PREPARE command from the global manager, the agent relays it to the local sites 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 sites.

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

The problem is that in a 3PC protocol one assumes that each site knows where the other sites are and can always reach them. While in our case each site indeed knows about the other sites that are in its *local* database, it does not know about sites that are in other participating databases, and thus cannot reach them during the termination protocol. In effect, by having the local sites communicate information to the global coordinator only through their agent, a single site failure (the agent's) is equivalent to a network partition. Therefore, although our failure model allows only site failures, we reach a situation which is similar to a communication failure, which 3PC does not tolerate (see [1]).

What needs to be done is to ensure that site failures are not equivalent to network partition. We can start by making sure that every site knows about every other participating site. However, this simple fix does not result in a working solution. For example, even if we assume that we can send in the EXEC statements the identities of all the participants (at all databases) that participate in a transaction that still would not solve our problem. We have a difficult translation problem in a number of ways: participants belonging to different databases might have different message formats, the same global transaction will result in different transaction ID's locally, the databases possibly use distinct election protocols, etc. Moreover, for remote sites, their local coordinator is still alive, and thus those participants are not ready to enter into an election protocol.

We now describe yet another modification to the 3PC global protocol just presented. We will place an auxiliary process at every site of every database (note that before we only had a single agent at one of the sites of the database). These auxiliary processes will act just like another site in the local distributed database, and will participate in the local commit protocol. But these auxiliary processes will actually be 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.

Our new protocol is very similar to the 3PC one presented above. In case of no failures it performs in exactly the same manner (other than the auxiliary processes being involved). Let us now consider what happens if one of the agents (of one 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 databases and auxiliaries what their state is, the databases will answer truthfully, but the auxiliaries may have to do some work before answering. Each auxiliary process may have to contact its *cohorts* (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 tells the local coordinator that it is in that state. If the auxiliary process is in the *wait* state, it must find out if there is another process in its cohort which is in a *prepare* state. If there is one, the auxiliary will react as though its state is *prepare*. This is done in order to guarantee that the local database will not decide to abort while other databases will decide to commit (since those databases have a process in a *prepare* state). 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 will tell the local coordinator that the auxiliary process state is *wait*, which will result in an abort decision by the local coordinator.

However, there is still a problem with this version of our protocol. Consider the following scenario. We have two 3PC databases. The global manager is in a *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 sites (which receive the message). Now the first agent dies. The first database processes (real and auxiliary) proceed to commit (since the auxiliaries are in the *prepare* state they agree to this), and then all of the processes 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 are dead). The second database proceeds to abort, and now the protocol has failed.

The problem with this protocol is that some of the auxiliary processes are not aware that other auxiliaries are in a *prepare* state. Thus, under failure conditions, some remote auxiliaries might have forced a previous commit.

Finally, we now present a global protocol that fulfills the correctness criteria we describe in Appendix C. Our global protocol is as follows. As before, we place an auxiliary process at every site of every database, and each auxiliary will act as a regular participant. However, there is one crucial difference with the protocol 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.) During local transaction processing the auxiliary processes will not have work to do. If there is a global transaction, the global manager will send an EXEC message to the agents. The agents will send a local EXEC to the database participants, as well as the local auxiliary processes. If they all reply OK, the agent will send an OK to the global manager. If the latter receives OK from all the agents, it will send a PREPARE_TO_PREPARE message to all the agents, who will then send an equivalent message to the auxiliaries (but not to the other participants). The auxiliaries will acknowledge the PREPARE_TO_PREPARE message, and once the agent receives their acknowledgements, it informs the global manager of that fact. The global manager will now send a PREPARE message to all the agents, who will forward it to all the local sites (including the auxiliaries). Everyone will send an ACK to the agent, who will forward it to the global manager. When ACKs are received from all agents, the global manager will issue a COMMIT message to all agents, who in turn will send COMMITs to every local site.

Consider now what would happen during a failure. If there is a global manager failure at any time, the agents and auxiliaries will follow the usual 3PC election algorithm to select a new global manager. If there is a failure on one of the local sites (other than the agent site) the auxiliaries will follow the local commit protocol. If the site that fails is the local agent's, the auxiliary processes have some extra work to do. We describe this extra work below.

In the case of agent failure, all the sites in the database (as well as all local auxiliaries) will enter into the usual 3PC election protocol to select the new local coordinator. At this point the election protocol selects 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. The difficulty lies in that now we 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.

Thus, when a non-auxiliary is chosen as coordinator, the protocol proceeds as follows. The auxiliary process at the site where the new coordinator resides becomes the new agent for the database. The local termination protocol is now started. 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 will be either a PREPARE_TO_PREPARE or an ABORT. (Remember, a global manager failure at any point will result in the election of a new one, so the agent will always eventually receive either one of the two messages mentioned). If an ABORT is received, the new agent tells the coordinator that its state was *wait*, which will force 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 will then send 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*). And therefore none of the other databases will abort this transaction.

If instead 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. It then proceeds to contact the local coordinator and tell it what its state really is.

2PC databases are easy to integrate into the above scheme. As we indicated in the previous section, there will also be 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 will have to look 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, the participant cannot be sure it was the last process alive. Only the auxiliaries are willing to tell others what the outcome of the transaction was.

In Appendix B we formalize some of the algorithms presented above.

A few comments follow. 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, since it typically depends on the speed of the available communication network, which varies from installation to installation.) In theory, we need an interval large enough to allow n global manager failures, where n is the number of sites in the system (we do not allow failed sites to again join an ongoing transaction). In practice, one would need to worry about only a reasonable number of failures.

Secondly, instead of having agents at every database, we could adopt a model where the global transaction manager plays the role of the agent for all the databases (we would still need the auxiliary processes). All sites would then get commands directly from the global manager. However, we did not use this model because, although simpler to implement, in it the global manager becomes a performance bottleneck.

Lastly, 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.

Theorem 2 *In the absence of total failures, our algorithm and its termination protocol never cause processes to block. (Recall that we consider 3PC databases only).*

Proof: It is easy to check by considering the three possible cases: no failures, a failure of global manager, a failure of an agent. After each failure the protocol continues until a decision is reached or all sites fail. □

We now prove correctness of the integrated commit protocol described above.

Recall that in [1] five requirements for correctness are presented (we also list them in Appendix C.6). Requirements 2-4 are obviously satisfied and we will not discuss them. It is straightforward to show that requirement 5 is also satisfied by considering the recovery mechanisms of the algorithm. Here we will only prove that requirement 1 also holds (i.e., all sites 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.

One final point before the proofs below. An important difference between our algorithm and the other commit algorithms is that a local coordinator might run to termination and commit/abort before (or after) the global manager has decided on it. Recall that when a local coordinator fails and a new one is selected, the new coordinator is not aware of taking part in a global transaction. Therefore, it executes the local termination algorithm independently of other databases or the global manager. We thus want to make sure that the decisions taken by the various local coordinators and that taken by the global manager are consistent. (We know that local processes and local coordinators will agree by the correctness of the local protocols.)

We will divide time into *intervals* between the elections of global managers. We will proceed in two steps. The first step (*theorem 4*) proves that all decisions taken within a specific interval are consistent. The second step (*theorem 7*) proves that decisions taken at any interval are consistent with those taken in earlier intervals. We will 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 3 *When a local coordinator decides to commit, its agent reported either* prepare *state or* commit *state. When a local coordinator decides to abort, its agent reported either* abort *state or* wait *state.*

Proof:

The proof will proceed by induction on the number of failures of the local coordinator (in a specific database).

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. We will 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.

Suppose now 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. 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 ABORT vote was sent to the global manager by an agent (while this agent was acting as local coordinator, i.e., before any coordinator failure), 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, i.e., the global manager intends to abort the transaction. 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 4 *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 3 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. □

Lemma 5 *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. □

Lemma 6 *Consider the i^{th} interval. 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. □

Theorem 7 *Under our protocol, all operational processes reach the same decision.*

Proof: The proof will proceed by induction on i, the i^{th} interval.

For $i = 0$ (before the first failure of the global manager): We know from Theorem 4 that all processes that reach a decision in an interval reach the same conclusion.

The induction step: By Theorem 4, 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 will 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 (by similar reasoning as in Lemma 3). 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 manager 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 some 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 3 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 5, no process can have previously committed.

Let us consider now the case where a decision is made to commit. If a decision to commit is taken because some operational process reports that it is in a *commit* state, it is again not difficult to see that an abort decision could not have been made earlier (by a similar argument as above). 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 6. Thus, the global manager could not have decided to abort, nor would it have instructed any agent to force a local abort. □

5 Summary

In this paper we have discussed the problem of atomic commitment in a system of heterogeneous distributed databases. We have examined two cases. The first case was that of non-externalized commit databases for which we only categorized some of the approaches described in the literature. The second case, which we have explored in detail, was that of externalized commit databases.

First, we presented a global commit protocol for heterogeneous databases composed of distributed databases which use any of the major types of two-phase commit protocols. We did this by having our global coordinator engage in a centralized two-phase commit with agent processes at the participating databases, and showing that we could integrate the various kinds of two-phase protocols into our global one. The advantages of our global protocol are its simplicity and that it addresses the most important case in practice, that of integrating two-phase commit databases.

We then considered database systems using either three-phase commit or two-phase commit protocols. In developing the global commit protocol for this case we required the use of helping processes (agents and auxiliary processes) at each local database. The overhead of such processes, as well as possibly the increased time-out intervals that our algorithms will probably require in a real implementation are the negative aspects of our proposed solutions. The main contribution of our work is that we have shown how to integrate a heterogeneous collection of databases which use most of the commonly studied commit protocols. Furthermore, our proposed commit 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 the Matsushita Information Technology Laboratory. 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.) Even if none of these sources use a standard version of 3PC, we hope that the work described in this paper will be of help in integrating other non-blocking protocols.

Acknowledgements

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

References

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

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

[3] W. Du and A.K. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. *Proceedings of VLDB*, p. 347-355, 1989.

[4] H. Garcia-Molina, K. Salem. SAGAS. *Proceedings ACM SIGMOD*, p. 249- 259, 1987.

[5] D. Georgakopoulos, M.Rusinkiewicz and A. Sheth. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. *Data Engineering* 91.

[6] 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.

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

[8] D. Heimbigner and D. McLeod. A Federated Architecture for Information Management. *Integration of Information Systems: Bridging Heterogeneous Databases* (see [7]), p. 46-71.

[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] W. Litwin and A. Abdellatif. Multidatabase Interoperability. *Integration of Information Systems: Bridging Heterogeneous Databases* (see [7]), p.213-220.

[12] C. Pu. Superdatabases for Composition of Heterogeneous Databases. *, Integration of Information Systems: Bridging Heterogeneous Databases* (see [7]), p. 150-157.

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

A Algorithm for Merging 2PC Databases

In this appendix we present the global commit algorithm of Section 3 from the point of view of the global manager and the agent. We also present the agent's recovery algorithm. All algorithms are presented in a Pascal-like language.

Global manager's algorithm

```
send EXEC to all agents;
write start-2pc record in the log file;
wait for vote messages (OK or NOK) from all agents
    on timeout begin
        write abort record in the log file;
        send ABORT to all agents from which OK was received;
    end;
if all votes were OK then begin
    write commit record in the log file;
    send commit to all agents;
end
else begin
    write abort record in the log file;
    send ABORT to all agents from which OK was received;
end;
return
```

Agent's algorithm

```
wait for EXEC from the global manager
    on timeout begin
        write abort record in the log file;
        return;
    end;
write EXEC record in the log file;
/* send EXEC to local participants
 * according to the local protocol */
case the local protocols is
    hierarchical 2PC:
        send EXEC to all local participants
```

```
        in the next level of the hierarchy;
    centralized 2PC:
        send EXEC to all local participants;
    decentralized 2PC:
        send OK to all local participants
        (and to the auxiliary process);
    linear 2PC:
        send OK to the first participant in the chain
        (the last process in the chain is the auxiliary process);
end; /* case */
write start-2pc record in the log file;
wait for vote messages (OK or NOK) from all
    relevant participants
    on timeout begin
        write abort record in the log file;
        send ABORT to all participants from which
        OK was received;
        send ABORT to the global manager;
        return;
    end;
if all votes were OK and the agent votes OK then begin
    write OK in the log file;
    send OK to the global manager;
    wait for decision (COMMIT or ABORT) from the
    global manager;
        on timeout initiate termination protocol;
        /* which is a loop of sending messages
         * DECISION_REQ to global manager and agents */
    if decision message is COMMIT then begin
        write commit record in the log file;
        /* send COMMIT to local participants
         * according to the local protocol */
        case the local protocols is
            hierarchical 2PC:
                send COMMIT to all local participants
                in the next level of the hierarchy;
            centralized 2PC:
                send COMMIT to all local participants;
            decentralized 2PC:
                send COMMIT to the auxiliary process;
                the auxiliary process will then send OK to
                    all local participants;
```

```
                linear 2PC:
                    send COMMIT to the auxiliary process;
                    the auxiliary process then sends the COMMIT
                        message to the next-to-last process in
                        the chain;
            end; /* case */
        end;
        else begin
            write abort record in the log file;
            send ABORT to all participants from which
            OK was received;
        end;
    end
    else  /* some participants' votes are NOK */ begin
        write abort record in the log file;
        send ABORT to all participants from which OK was received;
        send ABORT to the global manager;
    end;
```

Agent's recovery algorithm

```
    if either commit or abort record appears in the
    agent's log file then begin
        commit or abort (accordingly);
        return;
    end;
    LOOP: send DECISION-REQ to the global manager
        (and possibly to the other agents);
    wait for decision message from any of the above participants
        on timeout goto LOOP;
    if the decision message is COMMIT then begin
        write commit record in the log file;
        send COMMIT (or OK, depending upon the algorithm)
           to all relevant participants;
    end;
    else begin /* decision message is ABORT */
        write abort record in the log file;
        send ABORT to all relevant participants;
    end;
```

B Algorithm for Merging 2PC and 3PC Databases

In this appendix we present the global commit algorithm of Section 4 from the point of view of the agent and the auxiliary process. We also describe the termination protocol of the auxiliary process that resides at the site of the newly-elected coordinator after an agent failure.

Agent's algorithm

```
wait for EXEC from the global manager
    on timeout begin
        write abort record in the log file;
        return;
    end;
write EXEC in log file;
send EXEC to all local participants;
wait for vote message (OK or NOK) from all the
    local participants
    on timeout begin
        write ABORT in the log file;
        send ABORT to all processes from which OK was received;
        send ABORT to the global manager;
        return;
    end;
if all messages were OK then begin
    write OK in the log file;
    send OK to the global manager;
    wait for PREPARE_TO_PREPARE or ABORT from the global manager
        on timeout begin
            initiate global election protocol;
            if elected then invoke global manager's
                algorithm of termination protocol;
            else invoke agent's algorithm of
                termination algorithm;
            return;
        end;
    if message received is PREPARE_TO_PREPARE then begin
        write PREPARE_TO_PREPARE in the log file;
        send PREPARE_TO_PREPARE to all local auxiliaries;
        wait for ACK from all local auxiliaries;
            on timeout skip;
            /* ignore participant failures */
        send ACK to the global manager;
```

```
        wait for PREPARE from the global manager
            on timeout begin
                initiate global election protocol;
                if elected then invoke global manager's
                    algorithm of termination protocol;
                else invoke agent's algorithm of
                    termination algorithm;
                return;
            end;
        write PREPARE in the log file;
        send PREPARE to all local participants;
        wait for ACK from all local participants;
            on timeout skip;
            /* ignore participant failures */
        send ACK to the global manager;
        wait for COMMIT from the global manager
            on timeout begin
                initiate global election protocol;
                if elected then
                    invoke global manager's algorithm
                    of termination protocol;
                else invoke agent's algorithm of
                    termination protocol;
                return;
           end;
        write COMMIT record in the log file;
    end;
    else /* message is ABORT */ begin
        write ABORT in the log file;
        send ABORT to all local participants;
    end;
else /* some local process voted NOK */ begin
    write ABORT in the log file;
    send ABORT to all processes from which OK was received;
    send ABORT to the global manager;
end;
return;
```

Auxiliary's algorithm

```
wait for EXEC from the local agent in the database
    on timeout begin
        write ABORT record in the log file;
```

```
        return;
    end;
write EXEC in log file;
write OK in the log file;
send OK to the local coordinator;
wait for PREPARE_TO_PREPARE from agent or ABORT from the local coordinator;
    on timeout begin
       initiate local election protocol;
       if elected then invoke local agent's
          algorithm of termination protocol;
       else invoke auxiliary's algorithm of
          termination algorithm;
       return;
    end;
if message received is PREPARE_TO_PREPARE then begin
    write PREPARE_TO_PREPARE in the log file;
    send ACK to the local agent;
    wait for PREPARE from the local coordinator;
        on timeout begin
           initiate local election protocol;
           if elected then invoke local agent's
              algorithm of termination protocol;
           else invoke auxiliary's algorithm of
              termination algorithm;
           return;
        end;
    send ACK to the local agent;
end;
else /* an ABORT was received */ begin
     write ABORT record in the log file;
     return;
end;
wait for COMMIT from the local coordinator
    on timeout begin
         initiate local election protocol;
         if elected then
            invoke local agent's algorithm
              of termination protocol;
         else invoke auxiliary's algorithm of
              termination protocol;
         return;
     end;
```

```
write COMMIT record in the log file;
return;
```

New Agent's Termination Protocol

```
start: wait for STATE_REQ from the (new) local coordinator;
    on timeout begin
        initiate local election protocol;
        if elected then begin
            execute local coordinator's termination algorithm;
            return;
        end;
        else goto start;
    end;
if your state is ABORT, COMMIT or PREPARE then begin
    report your state;
    return;
end;
loop: send STATE_REQ to the global manager;
  wait for state report
    on timeout begin
        initiate global election protocol;
        if elected then begin
            execute global manager's termination algorithm;
            return;
        end;
        else goto loop;
    end;
/* report can be ABORT, PREPARE_TO_PREPARE or PREPARE */
if the state report message was ABORT then begin
    write abort record in the log file;
    send wait to the local coordinator;
end;
else if state report is PREPARE_TO_PREPARE then begin
    write prepare_to_prepare record in the log file;
    send PREPARE_TO_PREPARE to the local auxiliaries;
    wait for ACK from all auxiliaries
        on timeout skip; /* ignore site failures */
    end;
    send ACK to the global manager;
    wait for PREPARE from the global manager
        on timeout begin
            initiate global election protocol;
```

```
            if elected then begin
                execute global manager's termination algorithm;
                return;
            end;
            else goto loop;
    send PREPARE to the local coordinator;
else /* state report is PREPARE */ begin
    report PREPARE to the local coordinator
    return;
end;

end;
```

C 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. We also remind the reader of the correctness criteria for commit protocols. For more details on this subject see [1].

C.1 Centralized 2PC

Centralized 2PC is described in [6] and [10]. One of the sites 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 will decide to commit and will send the COMMIT message to all participants if all votes are OK and the coordinator does not timeout. Otherwise, the coordinator will send an ABORT message.
4. Each participant waits for the COMMIT or ABORT message and acts accordingly.

The sites 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.

C.2 Decentralized 2PC

The decentralized 2PC protocol is presented in [14]. All participating sites 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 sites can be in are the same as in the centralized version.

C.3 Linear 2PC

The linear 2PC protocol appears in [6] and [13]. Sites 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 site 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 sites can be in are the same as in the centralized 2PC case.

C.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 will decide to commit and will send the COMMIT message to all its children if all votes are OK and the coordinator does not timeout. Otherwise, the coordinator will send 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.

C.5 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 will decide to commit and will send the PREPARE message to all participants if all votes are OK and the coordinator does not timeout. Otherwise, the coordinator will send 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 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 will decide on abort.

2. If some process is committed, the coordinator will decide on commit.

3. If no process has received the PREPARE message, the coordinator will decide to abort.

4. If some process has received the PREPARE message, the coordinator will continue the protocol.

The sites can be in either one of the states mentioned above or in a 4th state called a *prepare* state (when it acknowledges the PREPARE message).

It should be noted that, in order to elect a new coordinator, each participants has to know the identities of the others. This knowledge is essential to our discussion. Either the coordinator can send the name of the participants in the EXEC message or it can be assumed that the participants know each other in advance.

C.6 Correctness Criteria

There are five requirements to be satisfied for an atomic commitment algorithm to be considered correct (see [1]):

1. All sites that reach a decision reach the same one.
2. Once made, a decision can not be reversed.
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 (but 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.