On the Heterogeneity of Distributed Databases
Integrating Commit Protocols

Constantinos V. Papadopoulos
Department of Computer Science, University of Piraeus, Greece
and Computing Division, Greek Ministry of Finance (KEPYO)

Abstract **This paper explores atomic commitment among heterogeneous distributed databases. Two cases are considered, one where the individual databases participating in the system do not externalize their commit protocols, and a second, where the participating databases do externalize their commit protocols. With regard to the latter case I am proposing new heterogeneous commit protocols, one for joining databases that have different 2-phase commit protocols, and one for uniting databases which employ a 3-phase protocol with others employing a 2-phase commit protocol.**

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

1. INTRODUCTION

Heterogeneous databases allow the physical and logical distribution of data. There exist environments where a collection of autonomous databases need to cooperate with each other despite their heterogeneity. Such environments have been discussed in various articles (e.g. [7]), still there are many open problems like semantic incompleteness and inconsistency, naming services, security, concurrency control, deadlock detection and atomic commitment.

This article addresses the problem of atomic commitment for autonomous heterogeneous databases. Considering the integration of autonomous databases, we must distinguish between two different kinds of systems. The first one consists of databases that externalize their commit protocols (*externalized commit databases*). These systems make public whether they use 2-phase commit, 3-phase commit, a version of the Byzantine agreement protocol etc. They also make available the internal state of the commit protocols (i.e. wait, prepare, 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 area of heterogeneous databases has addressed the latter case. The central issue of my article is atomic commitment for databases that do externalize the commit protocols.

The description of the integration problem for externalized commit databases will be accompanied in this article by some examples that show why it is not trivial to solve, as well as by novel protocols for heterogeneous commit. Two protocols are presented, the first dealing with environments where all the databases have some type of 2-phase commit protocol (i.e. centralized, linear, etc.), and the second concerning the more difficult case of integrating 2-phase commit (2PC) and 3-phase commit (3PC) protocols. The main advantage of 3-phase commit over its 2-phase equivalent is that it is a nonblocking protocol. Although 3PC has not been used extensively so far, it 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. Hence autonomy is preserved.

2. MERGING 2-PHASE COMMIT DATABASES

Although non-externalized commit databases may also be integrated, one should expect that some functionality would be sacrificed for this purpose. The integration of databases that do externalize their commit protocols can, instead, be implemented more efficiently. To this end, it is necessary to rely upon a generic database model which I describe next.

The Model : I assume my heterogeneous database will be built on top of existing distributed databases. A basic assumption of this model is that no modification is allowed to be made to the source code of the databases; it is mandatory that these are treated as "black boxes", except that their commit protocols are known in advance. A *local transaction* is defined as that transaction that resides in a single distributed database. If a local

transaction is executed in several sites, all those sites belong to a single logical database. In turn, a *global transaction* involves several distributed databases. Every global transaction is sent to the so-called *global transaction manager.* This 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 discussing global protocols in this section (and the next) I assume that the participating distributed databases use either the 3-phase commit protocol or a variant of the 2-phase commit one (being it centralized, decentralized, linear or hierarchical). Moreover, when discussing the 3PC protocol, a reliable network is assumed, because this protocol can tolerate only site failures. In the case of the 2PC protocol, I shall take into account communication failures as well.

Consider now a heterogeneous database composed of only 2PC databases. In fact, this is the most common case. There is less overhead involved in the solution to this case than in the protocol that I will propose for 3PC databases. The global protocol for integrating 2-phase commit protocols of multiple databases is the following :

Protocol : The global transaction manager will run a centralized 2PC protocol with the agents as participants. The global manager will play the role of coordinator. Each one of the possible versions of 2-phase commit is considered below along with the way that this local algorithm can be meshed with the global one.

(α) *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 it to the local sites. After collecting the answers from the local sites, the agent returns the answer to the global manager.

(β) *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 OKs 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 to take.

(γ) *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 tothe local sites, these 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). There are a number of ways of dealing with this problem, but I shall only mention one here. A posible 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 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.

(δ) *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 I again need the help of an auxiliary process, which I will force to be the rightmost participant in any commit that involves a global transaction. In other words, my 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 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 I have assumed that the information is not predetermined (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.

Theorem 1 *- The algorithm presented above meets the following correctness conditions :* **(i)** *all sites that reach a decision reach the same one,* **(ii)** *once made , a decision cannot be reversed,* **(iii)** *the commit decision can only be reached if all processes voted OK,* **(iv)** *if there are no failures and all processes voted OK, the decision will be to commit,* **(v)** *Given any execution schedule containing failures (from those that the algorithm is designed to tolerate), if all the failures are repaired and no new failures occur for a sufficiently long time, then all processes will eventually reach a decision.*

I prove this theorem by first showing that my modifications to the local commit protocols have not affected their correctness and then showing that the combined protocol is error-free. With regard to the fiest step of the proof, it is easy to see that my claim is true when the local 2PC is either hierarchical or centralized (because I 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 I 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 obeyall the rules of the protocol (the only difference between the auxiliary and the 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 my 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 of the global manager, the protocol is a hierarchical 2PC, since it is not aware of the internal variation of 2PC that 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 *1st)* getting instructions from the global manager and passing them to the local participants, and *2nd)* 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. Thus, by a reasoning similar to that of the correctness proof for hierarchical 2PC it is possible to show that the protocol is correct □

3. MERGING 2-PHASE COMMIT AND 3-PHASE COMMIT DATABASES

It is impossible to use a global 2PC protocol between the global manager and the agents, nor does the straightforward use of a 3PC protocol fullfil this need. Consider the following commit scenario: 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.

3.1 A MODIFIED VERSION OF THE 3PC PROTOCOL

I now describe yet another modification to the 3PC global protocol just presented. I shall place an auxiliary process at every site of every database (note that before I 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 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. My 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 me now bconsider 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 truthsfully, 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.

I am now presenting a global protocol that fulfills the correctness criteria described in Theorem 1. My global protocol is as follows. As before, I 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. Let me now describe in more detail the actions of my protocol when the local database follows a 3PC local protocol. (The 2PC case will be addressed 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.

3.2 PROOF OF CORRECTNESS OF THE INTEGRATED COMMIT PROTOCOL

To prove the correctness of the protocol, I take into consideration (once again) the correctness criteria of *Theorem 1* presented in section 2. Requirements (ii), (iii) and (iv) are obviously satisfied and I will not discuss them. Requirement (v) is also straightforward to prove by considering the recovery mechanisms of the algorithm. So it remains to prove that requirement (i) also holds.

The case of 2PC databases poses no difficulty. Clearly, there will not be a disagreement between two processes in the 2PC database by the correctness of 2PC. I 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 failure, and wait until it is prepared to continue. Below I prove that the global manager will always agree with 3PC processes. So, there is no possibility of a disagreement between a 2PC and a 3PC process. To simplify the proofs below, I now omit mention of 2PC databases in them.

I shall divide time into *intervals* between the elections of global managers. I shall 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. I shall consider a local coordinator as having committed at the *ith* interval if its agent has consulted with the *ith* global manager before reporting to the local coordinator. 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 occured, the local coordinator is the agent. Thus, it will decide to commit only if its state is *commit*, and likewise for an abort.

Phase 1 : 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. I shall look at the first phase in which any process in the database got a PREPARE message. According to my algorithm, at that time all auxiliaries had to be in at least a *prepare_to_prepare* state. Therefore, at the *ith* phase of 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 to a *prepare_to_prepare* state. This is so because either the ABORT vote was sent to the global manager by an agent, 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 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 *ith* agent, will not proceed to the *prepare_to_prepare* state. □

Theorem 4 *- All processes that reach a decision within the ith interval, reach the same one.*

Proof : At the *ith* interval all agents consult with the same global manager which will not report contradictory decisions. By Lemma 3 it is known 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). I shall 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 my 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 ith interval. If a process p that is operational during at least part of this time is in* prepare_to_prepare *state, then some some process q that was operational in (i-1)th interval was in* prepare_to_prepare *state then.*

Proof : At the beginning of the ith 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 the global commit protocol, all operational processes reach the same decision.*

Proof : The proof will proceed by induction on *i*, the *ith* interval.

For $i = 0$ (before the first failure of the global manager): I 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. I shall 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 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 reasonong 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. Therefore, none of the managers/coordinators could have sent an ABORT message since then. □

4. MERGING ALGORITHMS

In this section I am presenting the merging algorithms of the previous two sections in a Pascal-like form. I consider first the case of merging 2PC databases, where the global commit protocol is realized by the global manager and the various agents. The agent's recovery algorithm is presented as well.

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 protocol is
      hierarchical 2PC:
         send EXEC to all participants in the next level of the hierarchy;
      centralized 2PC:
         send EXEC to all 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 protocol is
               hierarchical 2PC:
                send COMMIT to all local participants in the next level of the hierarchy;
               centralized 2PC:
                  send COMMIT to all local mparticipants;
               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;
               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;
```

5. SUMMARY

In this article I have discussed the problem of atomic commitment in a system of heterogeneous distributed databases. In particular, I have examined the case of externalized commit databases. I presented first a global commit protocol for heterogeneous databases, composed of distributed databases which use any of the major types of 2-phase commit protocols. The advantages of this protocol are its simplicity and that it addresses the most important case in practice, that of integrating 2-phase commit databases. I then considered database systems using either 3-phase commit or 2-phase commit protocols. In developing the global commit protocol for this case I required the use of helping processes (agent and auxiliary ones) at each local database. The overhead of such processes, as well as the increased time-out intervals that my algorithms will probably require in a real implementation are the negative aspects of the proposed solutions. the main contribution of this work is that I have shown how to integrate a heterogeneous collection of databases which use most of the commonly studied commit protocols. Furthermore, the proposed commit protocol does not block a database because of remote failures, thus preserving local autonomy.

6. REFERENCES

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

[2] Y.Breitbart, A.Silberschatz, G. Thompson "Reliable Transaction Management in a Multidatabase System", *Proceedigs ACM SIGMOD,* 1990.

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

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

[5] D.Georgakopoulos, M.Rusinkiewicz, A.Sheth " On Serializability of Multidatabase Transactions through Forced Local Conflicts ", *Data Engineering,* 1991.

[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, D.McLeod " A Federated Architecture for Information Management ", *Integration of Information Systems:Bridging Heterogeneous Databases* (see [7]), pp.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, H.Sturgis " Crash Recovery in a Distributed Data Storage System ", *Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center,* 1976.

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

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

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

[14] D.Skeen " Nonblocking Commit Protocols "*Proceedings ACM SIGMOD Conference on Management of Data,* pp.133-147, June 1982.

[15] D.Skeen " A Quorum Based Commit Protocol ", *Proceedings 6th Berkeley Workshop on Distributed Data Management and Computer Networks, ACM/IEEE,* pp.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.