Chapter 17

REAL-TIME COMMIT PROCESSING

Jayant R. Haritsa

Database Systems Lab, Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, INDIA.

haritsa@dsl.serc.iisc.ernet.in

Krithi Ramamritham

Dept. of Computer Science and Engineering,
Indian Institute of Technology, Mumbai 400076, INDIA.

krithi@cse.iitb.ernet.in

Ramesh Gupta

Goldencom Technologies, Inc.
392 Acoma Way, Fremont, CA 94539, USA.

ramesh@goldencom.com

1. INTRODUCTION

Many real-time database applications are inherently *distributed* in nature [24, 28]. These include the intelligent network services database described in [5] and the mobile telecommunication system discussed in [29]. More recent applications include the multitude of directory, data-feed and electronic commerce services that have become available on the World Wide Web.

An essential requirement for most distributed real-time database system (**DRTDBS**) applications is *transaction atomicity* – this is satisfied by implementing a transaction *commit protocol*. Commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be "forced", that is, flushed to disk immediately in a synchronous manner. Due to this series of synchronous message and logging costs, commit processing can significantly increase the execution times of transactions [17, 7, 25]. This is especially problematic in the real-time context since it has a direct adverse effect on the system's ability to meet transaction timing constraints. Therefore, *the choice of commit protocol is an important design decision for DRTDBS*. Unfortunately, there is little research literature on this topic, making it difficult for DRTDBS designers to make informed choices. We address this lacuna here with a detailed quantitative evaluation whose results represent the first such work in the area.

Our study is carried out in the context of real-time applications with "firm-deadlines" [15] – for such applications, completing a transaction after its has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are considered to be worthless and are immediately "killed" – that is, aborted and permanently discarded from the RTDBS without being executed to completion. Accordingly, the performance metric is *KillPercent*, the steady-state percentage of killed transactions.[1]

2. PERFORMANCE FRAMEWORK

From a performance perspective, commit protocols can be compared with respect to the following issues:

1. **Effect on Normal Processing:** This refers to the extent to which the commit protocol affects the normal (no-failure) distributed transaction processing performance. That is, how expensive is it to provide atomicity using this protocol?

2. **Resilience to Failures:** When a failure occurs in a distributed system, ideally, transaction processing in the rest of the system should not be affected during the recovery of the failed component. With "blocking commit protocols", failures can lead to transaction processing grinding to a halt (as explained in Section 3.), whereas "non-blocking protocols" prevent such disruptions by incurring *additional* messages and forced-log-writes.

3. **Speed of Recovery:** This refers to the time required for the database to be recovered to a consistent state when the failed site comes back up after a crash. That is, how long does it take before transaction processing can commence again in a recovering site?

[1] Or, equivalently, the percentage of missed deadlines.

Of the three issues highlighted above, the design emphasis of most commit protocols has been on the first two (effect on normal processing and resilience to failures) since they directly affect ongoing transaction processing. In comparison, the last issue (speed of recovery) appears less critical for two reasons: First, failure durations are usually orders of magnitude larger than recovery times. Second, failures are usually rare enough that we do not expect to see a difference in *average* performance among the protocols because of one commit protocol having a faster recovery time than the other. Based on this viewpoint, our focus here is on the *mechanisms* required during normal (no-failure) operation to provide for recoverability and resilience to failures, and not on the post-failure *recovery process*.

3. TRADITIONAL DISTRIBUTED COMMIT PROTOCOLS

We adopt the common "subtransaction model" [3] of distributed transaction execution in our study. In this model, there is one process, called the *master*, which is executed at the site where the transaction is submitted, and a set of other processes, called *cohorts*, which execute on behalf of the transaction at the various sites that are accessed by the transaction. Cohorts are created by the master sending a STARTWORK message to the local transaction manager at that site. This message includes the work to be done at that site and is passed on to the cohort. Each cohort sends a WORKDONE message to the master after it has completed its assigned data processing work, and the master initiates the commit protocol (only) after it has received this message from all its cohorts.

For the above transaction execution model, a variety of commit protocols have been devised, most of which are based on the classical **two-phase commit (2PC)** protocol [10]. In this section, we briefly describe the 2PC protocol and a few of its popular variants – complete descriptions are available in [20, 7, 23]. For ease of exposition, the following notation is used in the sequel – "SMALL CAPS FONT" for messages, "`typewriter font`" for log records, and "sans serif font" for transaction states.

3.1 TWO PHASE COMMIT

In the 2PC protocol, the master initiates the first phase of the commit protocol by sending PREPARE (to commit) messages in parallel to all its cohorts. Each cohort that is ready to commit first force-writes a `prepare` log record to its local stable storage and then sends a YES vote to the master. At this stage, the cohort has entered a prepared state wherein it cannot unilaterally commit or abort the transaction but has to wait for the final decision from the master. On the other hand, each cohort that decides to abort force-writes an `abort` log record and sends a NO vote to the master. Since a NO vote acts like a veto, the cohort is permitted to unilaterally abort the transaction without waiting for the decision from the master.

After the master receives votes from all its cohorts, the second phase of the protocol is initiated. If all the votes are YES, the master moves to a committing state by force-writing a `commit` log record and sending COMMIT messages to all its cohorts. Each cohort, upon receiving the COMMIT message, moves to the committing state, force-writes a `commit` log record, and sends an ACK message to the master.

On the other hand, if the master receives even one NO vote, it moves to the aborting state by force-writing an `abort` log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to the aborting state, force-write an `abort` log record and send an ACK message to the master.

Finally, the master, after receiving ACKs from all the prepared cohorts, writes an `end` log record and then "forgets" the transaction (by removing from virtual memory all information associated with the transaction).

3.2 PRESUMED ABORT AND PRESUMED COMMIT

Two variants of the 2PC protocol called **presumed abort (PA)** and **presumed commit (PC)** were presented in [20]. These protocols try to reduce the message and logging overheads by requiring all participating cohorts to follow certain rules at failure recovery time. The protocols have been implemented in a number of database products and PA is, in fact, now part of the ISO-OSI and X/OPEN distributed transaction processing standards [7].

3.3 THREE PHASE COMMIT

A fundamental problem with all of the above protocols is that cohorts may become *blocked* in the event of a site failure and remain blocked until the failed site recovers. To address the blocking problem, a **three phase commit (3PC)** protocol was proposed in [23]. This protocol achieves a non-blocking capability by inserting an extra phase, called the "precommit phase", in between the two phases of the 2PC protocol. In the precommit phase, a preliminary decision is reached regarding the fate of the transaction. The information made available to the participating sites as a result of this preliminary decision allows a global decision to be made despite a subsequent failure of the master site. Note, however, that the price of gaining non-blocking functionality is an increase in the communication and logging overheads since: (a) there is an extra round of message exchange between the master and the cohorts, and (b) both the master and the cohorts have to force-write additional log records in the precommit phase.

4. INADEQUACIES IN THE DRTDBS ENVIRONMENT

The commit protocols described in the previous section are designed for traditional database systems where transaction throughput or average response time is usually the primary performance metric. With respect to meeting (firm) real-time objectives, however, they fail on two related counts: First, by making prepared data inaccessible, they increase transaction blocking times and therefore have an adverse impact on the number of killed transactions. Second, *prioritized* scheduling policies are typically used in RTDBS to minimize the number of killed transactions. These commit protocols, however, do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [22]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable.

Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priorities. Removing priority inversion in the commit protocol, however, is *not fully feasible*. This is because, once a cohort reaches the prepared state, it has to retain all its *update* data locks until it receives the global decision from the master – this retention is *fundamentally* necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a “prepared cohort” of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a DRTDBS is *inherently susceptible to priority inversion*. More importantly, the priority inversion is *not bounded* since the time duration that a cohort is in the prepared state can be arbitrarily long (for example, due to network delays). If the inversion period is large, it may have a significant negative effect on performance.

It is important to note that this “prepared data blocking” is *distinct* from the “decision blocking” (because of failures) that was discussed in Section 3.. That is, in all the commit protocols, *including 3PC*, transactions can be affected by prepared data blocking. In fact, 3PC’s strategy for removing decision blocking *increases* the duration of prepared data blocking. Moreover, such data blocking occurs during normal processing whereas decision blocking only occurs during failure situations.

To address the above-mentioned drawbacks (prepared data inaccessibility and priority inversion) of the classical commit protocols, we have designed a new protocol called **PROMPT**. The PROMPT design is based on a specific semantics of firm deadlines in DRTDBS, defined in the following section – the description of PROMPT itself is deferred to Section 6..

5. FIRM DEADLINE SEMANTICS IN DRTDBS

The semantics of firm deadlines is that a transaction should be either committed before its deadline or be killed when the deadline expires. To implement this notion in a distributed RTDBS, ideally the *master and all the cohorts* of a successfully executed transaction should commit the transaction before the deadline expires or all of them should abort immediately upon deadline expiry. In practice, however, it is impossible to provide such guarantees because of the arbitrary message delays and the possibility of failures [4]. To avoid inconsistencies in such cases, we define the firm deadline semantics in the distributed environment as follows:

Definition: A distributed firm-deadline real-time transaction is said to be committed if the *master* has reached the commit decision (that is, forced the commit log record to the disk) before the expiry of the deadline at its site. This definition applies *irrespective of whether the cohorts have also received and recorded the commit decision by the deadline*.

To ensure transaction atomicity with the above definition, we require prepared cohorts that receive the final decision *after* the local expiry of the deadline to still implement this decision. Note that this is consistent with the intuitive notion of firm deadlines since all that happens is that access to prepared data (at cohort's site) is prevented even beyond the deadline until the decision is received by the cohort; other transactions which would normally expect the data to be released by the deadline only experience a delay. We expect that many real-time database applications, especially those related to electronic commerce (e.g., electronic auctions), will subscribe to these semantics.

Typically, the master is responsible for returning the results of a transaction to the invoker of the transaction. From the above discussion, it is clear that the semantics we prescribe are such that, if a transaction commits, its results will begin to be output *before* the deadline. Further, the problem of delayed access to data, even after the expiry of the deadline of the cohort holding these data items, applies primarily to the *classical protocols* – the effect is considerably reduced with PROMPT, as discussed in the following section.

6. THE PROMPT REAL-TIME COMMIT PROTOCOL

The main feature of our new **PROMPT** (Permits Reading Of Modified Prepared-data for Timeliness) commit protocol is that transactions requesting data items held by other transactions in the prepared state are allowed to access this data.That is, prepared cohorts *lend* their uncommitted data to concurrently executing transactions (without, of course, releasing the update locks). The mechanics of the interactions between such "lenders" and their associated "borrowers" are captured in the following three scenarios, only one of which will occur for each lending:

1. **Lender Receives Decision Before Borrower Completes Data Processing:** Here, the lending cohort receives its global decision before the borrowing cohort has completed its local data processing. If the global decision is to commit, the lending cohort completes in the normal fashion. On the other hand, if the global decision is to abort, the lender is aborted in the normal fashion. In addition, the borrower is also aborted since it has utilized dirty data.

2. **Borrower Completes Data Processing Before Lender Receives Decision:** Here, the borrowing cohort completes its local data processing before the lending cohort has received its global decision. The borrower is now "put on the shelf", that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the (commit-related) processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until either the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, the borrower is "taken off the shelf" (if it has no other "pending" lenders) and allowed to send its WORKDONE message, whereas if the lender aborts, the borrower is also aborted immediately since it has utilized dirty data (as in Scenario 1 above). In the latter case (deadline expiry), the borrower is killed in the normal manner.

3. **Borrower Aborts During Data Processing Before Lender Receives Decision:** Here, the borrowing cohort aborts in the course of its data processing (due to either a local problem, deadline expiry or receipt of an ABORT message from its master) before the lending cohort has received its global decision. In this situation, the borrower's updates are undone and the lending is nullified.

In summary, the PROMPT protocol allows transactions to access uncommitted data held by prepared transactions in the "optimistic" belief that this data will eventually be committed.[2] It uses this approach to mitigate the effects of both the data inaccessibility and the priority inversion problems that were identified earlier for traditional commit protocols (Section 4.).

We wish to clarify here that while the PROMPT design may superficially appear similar to that of optimistic concurrency control [16], it is actually quite different since updates are made in-place and not to copies or versions of the data; also, data is lent only by transactions that have completed their data processing and are in the prepared state.

6.1 ADDITIONAL REAL-TIME FEATURES OF PROMPT

To further improve its real-time performance, three additional features are included in the PROMPT protocol: Active Abort, Silent Kill and Healthy Lending. These features are described below.

6.1.1 Active Abort. In the basic 2PC protocol, cohorts are "passive" in that they inform the master of their status only upon explicit request by the master. This is acceptable in conventional distributed DBMS since, after a cohort has completed its data phase, there is *no possibility* of the cohort subsequently being aborted due to serializability considerations (assuming a locking-based concurrency control mechanism).

In a DRTDBS, however, a cohort which is not yet in its commit phase can be aborted due to conflicts with higher priority transactions. Therefore, it may be better for an aborting cohort to immediately inform the master so that the abort of the transaction at the sibling sites can be done earlier. Early restarts are beneficial in two ways: First, they provide more time for the restarted transaction to complete before its deadline. Second, they minimize the wastage of both logical and physical system resources. Accordingly, cohorts in PROMPT follow an "active abort" policy – they inform the master as soon as they decide to abort locally; the subsequent abort process implemented by the master is the same as that followed in the traditional passive environment.

6.1.2 Silent Kill. For a transaction that is killed before the master enters its commit phase, there is no need for the master to invoke the abort protocol since the cohorts of the transaction can *independently* realize the missing of the

[2]A similar, but unrelated, strategy of allowing access to uncommitted data has also been used to improve real-time *concurrency control* performance [2].

deadline (assuming global clock synchronization)[3]. Eliminating this round of messages may help to save system resources. Therefore, in PROMPT, aborts due to deadline misses that occur before the master has initiated the commit protocol are implemented "silently" without requiring any communication between the master and the cohort.

6.1.3 Healthy Lending. A committing transaction that is close to its deadline may be killed due to deadline expiry before its commit processing is over. Lendings by such transactions must be avoided since they are likely to result in the aborts of all the associated borrowers. To address this issue, we have added a feature to PROMPT whereby only "healthy" transactions, that is, transactions whose deadlines are not very close, are allowed to lend their prepared data. This is realized in the following manner: A *health factor*, HF_T, is associated with each transaction T and a transaction is allowed to lend its data only if its health factor is greater than a (system-specified) minimum value $MinHF$. The health factor is computed at the point of time when the master is ready to send the PREPARE messages and is defined to be the ratio *TimeLeft / MinTime*, where *TimeLeft* is the time left until the transaction's deadline, and *MinTime* is the minimum time required for commit processing (recall that a minimum of two messages and one force-write need to be processed before the master can take a decision).

The success of the above scheme is directly dependent on the threshold health factor $MinHF$ – set too conservatively (large values), it will turn off the borrowing feature to a large extent, thus effectively reducing PROMPT to standard 2PC; on the other hand, set too aggressively (small values), it will fail to stop several lenders that will eventually abort. In our experiments, we consider a range of values for $MinHF$ to determine the best choices.

An important point to note here is that the health factor is not used to decide the *fate* of the transaction but merely to decide *whether* the transaction can lend its data. Thus, erroneous estimates about the message processing times and log force-write times only affect the extent to which the optimistic feature of PROMPT is used, as explained above.

[3]Our firm deadline semantics ensure that skew in clock synchronization, if any, only affects performance, but not atomicity. Further, for minor skews, the performance impact is expected to be marginal.

6.2 ABORTS IN PROMPT DO NOT ARBITRARILY CASCADE

An important point to note here is that PROMPT's policy of using uncommitted data is generally *not recommended* in traditional database systems since this can potentially lead to the well-known problem of *cascading aborts* [1] if the transaction whose dirty data has been accessed is later aborted. However, for the PROMPT protocol, this problem is alleviated due to the following two reasons:

First, the lending transaction is *typically expected to commit* because: (a) The lending cohort is in the prepared state and cannot be aborted due to local data conflicts, and (b) The sibling cohorts are also expected to eventually vote to commit since they have survived[4] all their data conflicts that occurred prior to the initiation of the commit protocol (given our Active Abort policy).

The only situation where a lending cohort will finally abort is if (a) the deadline expires at the master's node before the master reaches a decision, or (b) a sibling cohort votes NO. The latter case can happen only if the ABORT message sent by the sibling cohort and the PREPARE message sent by the master to the sibling cohort "cross each other" on the network. As the time during which a message is in transit is usually small compared to the transaction execution times, these situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.[5]

Second, even if the lending transaction does eventually abort, it only results in the abort of the immediate borrower and does not cascade beyond this point (since the borrower is not in the prepared state, the only situation in which uncommitted data can be accessed). That is, a *borrower cannot simultaneously be a lender*. Therefore, the abort chain is bounded and is of length one. Of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the length of each abort *chain* is limited to one. In short, PROMPT implements a *controlled* lending policy.

6.3 SYSTEM INTEGRATION

To incorporate the PROMPT protocol in a DRTDBS, a few modifications are required to the standard transaction and lock manager programs. The important point to note about these modifications, which are described in detail in [14],

[4]We assume a locking-based concurrency control mechanism.

[5]Aborts could also occur after receiving the PREPARE message due to reasons such as violation of integrity constraints but our experiments have shown that unless the frequency of such aborts is unrealistically high, the improvement offered by PROMPT continues to be significant.

is that they are *local to each site* and do not require inter-site communication or coordination. Further, they do not appear difficult to incorporate in current database system software. In fact, some of them are *already provided* in current DBMS – for example, the high-performance industrial-strength ARIES recovery system [6] implements *operation logging* to support semantically rich lock modes that permit updating of uncommitted data. Moreover, as shown by our experiments, the performance benefits that can be derived from these changes more than compensate for the small amount of run-time overheads entailed by the above modifications and the effort needed to implement them.

6.4 INTEGRATING PROMPT WITH OTHER 2PC OPTIMIZATIONS

A particularly attractive feature of PROMPT is that it can be *integrated* with many of the other optimizations suggested for 2PC. For example, Presumed Commit and Presumed Abort (Section 3.) can be directly added as a useful supplement to reduce processing overheads. Moreover, the integration may often be *synergistic* in that PROMPT may retain the good features of the added optimization and simultaneously minimize its drawbacks. This is the case, for example, when PROMPT is combined with 3PC: In its attempt to prevent decision blocking, 3PC suffers an *increase* in the prepared data blocking period, but this drawback is reduced by PROMPT's lending feature.

Among additional optimizations [7], PROMPT can be integrated in a straightforward manner with *Read-Only* (one phase commit for read-only transactions), *Long Locks* (cohorts piggyback their commit acknowledgments onto subsequent messages to reduce network traffic), and *Shared Logs* (cohorts that execute at the same site as their master share the same log and therefore do not need to force-write their log records). Further, PROMPT is especially attractive to integrate with protocols such as *Group Commit* [7] (forced writes are batched together to save on disk I/O) and *linear 2PC* [10] (message overheads are reduced by ordering the sites in a linear chain for communication purposes). This is because these optimizations *extend*, like 3PC, the period during which data is held in the prepared state, thereby allowing PROMPT to play a greater role in improving system performance.

7. THE PIC PROTOCOL

A plausible alternative approach to that of PROMPT is to use the **priority inheritance (PI)** mechanism [22]. Here, low priority transactions that block a high priority transaction inherit the priority of the high priority transaction. The expectation is that the blocking time of the high priority transaction will be reduced since the low priority transactions will now execute faster and therefore release their locks earlier.

A positive feature of the PI approach is that it *does not* run the risk of transaction aborts, unlike the lending approach. Further, a study of PI in the context of (centralized) transaction concurrency control [8] suggested that priority inheritance is useful only if it occurs towards the end of the low priority transaction's lifetime. This seems to fit well with handling priority inversion during commit processing since this stage occurs at the end of transaction execution.

Motivated by these considerations we now describe **Priority Inheritance Commit (PIC)**, a real-time commit protocol based on the PI approach. In the PIC protocol, when a high priority transaction is blocked due to the data locked by a low priority cohort in the prepared state, the latter inherits the priority of the former to expedite its commit processing. To propagate this inherited priority to the master and the sibling cohorts, the priority inheriting cohort sends a PRIORITY-INHERIT message to the master. The master, in turn, sends this message to all other cohorts. After the master or a cohort receives a PRIORITY-INHERIT message, all further processing related to the transaction at that site (processing of the messages, writing log records, etc.) is carried out at the inherited priority.

8. PERFORMANCE RESULTS

Using a detailed simulation model of a firm-deadline DRTDBS, we evaluated the performance of the various commit protocols described in the previous sections. Our experiments, which covered a variety of transaction workloads and system configurations, demonstrated the following:

1. Distributed commit processing can have considerably more effect than distributed data processing on the real-time performance, especially on systems with slow network interfaces. This highlights the need for developing commit protocols tuned to the real-time domain.

2. The classical commit protocols generally perform poorly in the real-time environment due to their passive nature and due to preventing access to data held by cohorts in the prepared state.

3. Our new protocol, PROMPT, provides significantly improved performance over the standard algorithms. Its good performance is attained primarily due to its optimistic borrowing of uncommitted data and Active Abort policy. The optimistic access significantly reduces the effect of priority inversion which is inevitable in the prepared state. Support-

ing statistics showed that PROMPT's optimism about uncommitted data is justified, especially under normal loads. The other optimizations of Silent Kill and Presumed Commit/Abort, however, had comparatively little beneficial effect.

4. By appropriately setting the $MinHF$ parameter, most of the unhealthy lendings in PROMPT can be eliminated. In particular, a threshold of $MinHF = 1$ provided good performance in all the scenarios considered in our study.

5. The gap between PROMPT's performance and that of the standard algorithms widens with increased data contention due to the corresponding increase in borrowing opportunities. This feature is especially important since while resource contention can usually be addressed by purchasing more and/or faster resources, there do not exist any equally simple mechanisms to reduce data contention.

6. Experiments combining PROMPT with 3PC indicate that the nonblocking functionality can be obtained in the real-time environment at a relatively modest cost in normal processing performance. This is especially encouraging given the high desirability of the nonblocking feature in the real-time environment.

7. PIC's performance is *virtually identical* to that of 2PC. The reason is that priority inheritance comes into play only when a high priority transaction is blocked by a low priority prepared cohort, which means that this cohort has already sent the YES vote to its master. Since it takes *two* message delays for dissemination of the priority inheritance information to the sibling cohorts, PIC expedites at most the processing of *only the decision message*. Further, even the minor advantage that may be obtained by PIC is partially offset by the extra overheads involved in processing the priority inheritance information messages.

8. Finally, the difference between the performance of PROMPT (with $MinHF = 1$) and that of Shadow-PROMPT – an artificially constructed protocol, based on the "shadow transaction" approach of [2], that represents the *best on-line* usage of the optimistic lending approach – never exceeded *two percent*.

In summary, we suggest that DRTDBS designers may find the PROMPT protocol to be a good choice for high-performance real-time distributed commit processing. Viewed in toto, PROMPT is a portable (can be used with many other optimizations), practical (easy to implement), high-performance (substantially improves real-time performance) and efficient (makes good use of the lending approach) distributed commit protocol.

9. RELATED WORK

The material discussed in this chapter is based on our earlier papers [11, 12, 13, 14]. Apart from this, the only other work that we are aware of on DRTDBS commit processing is [4, 9, 29]. In this section, we briefly summarize these papers and contrast them with our study.

A *centralized timed* 2PC protocol is described in [4] that guarantees that the fate of a transaction (commit or abort) is known to all the cohorts before the expiry of the deadline when there are *no processor, communication or clock faults*. In case of faults, however, it is not possible to provide such guarantees, and an *exception state* is allowed which indicates the violation of the deadline. Further, the protocol assumes that it is possible for the DRTDBS to guarantee allocation of resources for a duration of time within a given time interval. Finally, the protocol is predicated upon the knowledge of worst-case communication delays.

Our work is different from [4] in the following respects: First, their deadline semantics are different from ours in that even if the coordinator of a transaction is able to reach the decision by the deadline, but it is not possible to convey the decision to all the cohorts by the deadline, the transaction is killed. Second, we do not assume any guarantees provided by the system for the services offered whereas such guarantees are fundamental to the design of their protocol. Note that in a dynamic prioritized system such guarantees are difficult to provide and, further, are generally not recommended since it requires *pre-allocation* of resources, thereby running the risk of priority inversion.

A common theme of allowing individual sites to *unilaterally* commit is used in [9, 29] – the idea is that unilateral commitment results in greater timeliness of actions. If it is later found that the decision is not consistent globally, "compensation" transactions are executed to rectify the errors. While the compensation-based approach certainly appears to have the potential to improve timeliness, there are quite a few practical difficulties. First, the standard notion of transaction atomicity is not supported – instead, a "relaxed" notion of atomicity [18] is provided. Second, the design of a compensating transaction is an application-specific task since it is based on the application semantics. Third, the compensation transactions need to be designed in advance so that they can be executed as soon as errors are detected – this means that the transaction workload must be fully characterized *a priori*. Finally, from a performance viewpoint also, there are some difficulties: (a) The execution of compensation transactions imposes an additional burden on the system; (b) It is not clear how the database system should schedule compensation transactions relative to normal transactions.

10. SUMMARY AND FUTURE WORK

In this chapter, we have attempted to characterize the state-of-the-art with regard to real-time commit processing. The few papers on this topic require fundamental alterations of the standard distributed DBMS framework. We have instead taken a different approach of achieving high-performance by incorporating novel protocol features.

We first precisely defined the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a distributed firm real-time setting. Subsequently, we evaluated the performance of a variety of standard commit protocols, as well as PROMPT, a new protocol that is designed specifically for the real-time environment and includes features such as controlled optimistic access to uncommitted data, Active Abort, Silent Kill and Healthy-Lending. Our performance results showed that PROMPT provides significantly improved performance over the classical commit protocols, and makes extremely efficient use of the lending premise. Finally, we also evaluated the priority inheritance approach to addressing the priority inversion problem associated with prepared data, but found it to be inherently unsuitable for the distributed environment.

We now outline a variety of issues that appear appropriate for further research: First, in all our experiments, update-oriented transaction workloads were modeled. It would be instructive to evaluate the performance for different mixes of read-only and update-oriented transaction workloads. Further, only a two-level transaction tree was considered in our work. The effects of the commit protocols on the performance of a system with “deep-transactions” (tree of processes architecture [19]) needs to be also evaluated.

Second, we assumed that all transactions have the same “criticality” or “value” [26], and therefore the performance goal was to minimize the *number* of killed transactions. An important open area is the characterization of commit protocol behavior for workloads where transactions have varying values, and the goal of the RTDBS is to maximize the *total value* of the completed transactions. These results could also be extended to “soft- deadline” applications, where there is a diminishing residual value for completing transactions even after their deadlines have expired.

Finally, our study was limited to commit protocols operating in two or three phases. A variety of *one-phase* protocols have also been proposed in the literature. In these protocols, cohorts enter the prepared state at the time of sending the WORKDONE message itself, thereby eliminating the voting phase and retaining only the decision phase. Examples of one-phase protocols include the Unsolicited Vote [27] protocol used in the Distributed Ingres database system and the Early Prepare [21] protocol.

The main attraction of one-phase protocols is that they significantly reduce the overheads of commit processing due to eliminating an entire phase. This reduction makes them especially attractive in the real-time environment. On the flip side, however, they significantly increase the amount of priority inversion due to the fact that cohorts are in a prepared state for extended periods of time. Therefore, a thorough evaluation needs to be made to assess the real-time capabilities of these protocols.

Acknowledgements Ramesh Gupta worked on this project when he was a graduate student at the Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India. S. Seshadri of Indian Institute of Technology, Mumbai, India, participated in the initial research efforts. Jayant Haritsa's work was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India. Krithi Ramamritham's work was supported in part by the US National Science Foundation Grant #IRI-9619588.

References

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

[2] A. Bestavros and S. Braoudakis, "Timeliness via Speculation for Real-Time Databases," in *Proc. of 15th Real-Time Systems Symp.*, December 1994.

[3] M. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," in *Proc. of 14th Intl. Conf. on Very Large Data Bases*, August 1988.

[4] S. Davidson, I. Lee, and V. Wolfe, "A Protocol for Timed Atomic Commitment," in *Proc. of 9th Intl. Conf. on Distributed Computing Systems*, 1989.

[5] B. Purimetla et al, "A Study of Distributed Real-Time Active Database Applications," in *IEEE Workshop on Parallel and Distributed Real-Time Systems*, 1993.

[6] C. Mohan et al, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging," *ACM Trans. on Database Systems*, vol. 17, no. 1, 1992.

[7] G. Samaras et al, "Two-Phase Commit Optimizations in a Commercial Distributed Environment," *Journal of Distributed and Parallel Databases*, vol. 3, no. 4, 1995.

[8] J. Huang et al, "On Using Priority Inheritance in Real-Time Databases," *Intl. Journal of Real-Time Systems*, vol. 4, no. 3, 1992.

[9] N. Soparkar et al, "Adaptive Commitment for Real-Time Distributed Transactions," Tech. Rep. TR-92-15, Dept. of Computer Science, Univ. of Texas-Austin, 1992.

[10] J. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, vol. 60 of *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[11] R. Gupta and J. Haritsa, "Commit Processing in Distributed Real-Time Database Systems," in *Proc. of National Conf. on Software for Real-Time Systems, Cochin, India*, January 1996.

[12] R. Gupta, J. Haritsa and K. Ramamritham, "More Optimism about Real-Time Distributed Commit Processing," in *Proc. of 18th IEEE Real-Time Systems Symp.*, December 1997.

[13] R. Gupta, J. Haritsa, K. Ramamritham and S. Seshadri, "Commit Processing in Distributed Real-Time Database Systems," in *Proc. of 17th IEEE Real-Time Systems Symp.*, December 1996.

[14] J. Haritsa, K. Ramamritham and R. Gupta, "The PROMPT Real-Time Commit Protocol," *IEEE Trans. on Parallel and Distributed Systems*, 2000 (in press).

[15] J. Haritsa, M. Carey and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *Intl. Journal of Real-Time Systems*, vol. 4, no. 3, 1992.

[16] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, vol. 6, no. 2, June 1981.

[17] B. Lampson and D. Lomet, "A New Presumed Commit Optimization for Two Phase Commit," in *Proc. of 19th Intl. Conf. on Very Large Data Bases*, August 1993.

[18] E. Levy, H. Korth and A. Silberschatz, "An Optimistic Commit Protocol for Distributed Transaction Management," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1991.

[19] B. Lindsay et al, "Computation and Communication in R^*: A Distributed Database Manager," *ACM Trans. on Computer Systems*, vol. 2, no. 1, 1984.

[20] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R^* Distributed Database Management System," *ACM Trans. on Database Systems*, vol. 11, no. 4, 1986.

[21] J. Stamos and F. Cristian, "A Low-Cost Atomic Commit Protocol", *Proc. of 9th Symp. on Reliable Distributed Systems*, October 1990.

[22] L. Sha, R. Rajkumar and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Tech. Rep. CMU-CS-87-181, Dept. of Computer Science, Carnegie Mellon Univ., 1987.

[23] D. Skeen, "Nonblocking Commit Protocols," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1981.

[24] S. Son, "Real-Time Database systems: A New Challenge," *Data Engineering Bulletin*, vol. 13, no. 4, December 1990.

[25] P. Spiro, A. Joshi and T. Rengarajan, "Designing an Optimized Transaction Commit Protocol," *Digital Technical Journal*, vol. 3, no. 1, 1991.

[26] J. Stankovic and W. Zhao. On Real-Time Transactions. *ACM SIGMOD Record*, March 1988.

[27] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres," *IEEE Trans. on Software Engg.*, vol. 5, no. 3, 1979.

[28] O. Ulusoy, "Research Issues in Real-Time Database Systems," Tech. Rep. BU-CEIS-94-32, Dept. of Computer Engg. and Information Science, Bilkent Univ., Turkey, 1994.

[29] Y. Yoon, *Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems*, Ph.D. thesis, Korea Advanced Institute of Science and Technology, May 1994.

Chapter 18

MOBILE DISTRIBUTED REAL-TIME DATABASE SYSTEMS

Kam-Yiu Lam

Department of Computer Science
City University of Hong Kong, Kowloon, Hong Kong

cskylam@cityu.edu.hk

Tei-Wei Kuo

Department of Computer Science and Information Engineering
National Taiwan University

ktw@csie.ntu.edu.tw

1. INTRODUCTION

Recent advances in mobile communication technology have made mobile computing a reality [5, 6, 12]. Various novel mobile computing applications, such as tele-medicine systems, real-time traffic information and navigation systems, and mobile Internet stock trading systems, are emerging as mobile users require instant access to information using their palmtops, personal digital assistant (PDA) and notebook computers not matter where they are. The realization of "instant" information access over a mobile network relies on real-time processing of transactions. As a result, the research on concurrency control and transaction scheduling for *mobile distributed real-time database systems* (MDRTDBS) is receiving growing attention in recent years [2, 8, 15, 17].

Although many researchers, e.g., [1, 3, 13, 16], have done excellent research in concurrency control and transaction scheduling for single-site and distributed RTDBS, not much work has been done for MDRTDBS. Owing to the intrinsic limitations of mobile computing systems, such as limited bandwidth and frequent disconnection, the efficient techniques for *distributed real-time database* *systems* (DRTDBS) which are supported by wired networks, may not be suitable and useful to MDRTDBS. Compared to wired networks, mobile networks are much slower, unreliable, and unpredictable. The mobility of clients affects the distribution of workload in the system. Disconnection between the mobile clients and the base stations is common [5]. These unique features of the systems can seriously affect the probability of data conflicts and deadline missing probability. Furthermore, overheads in resolving data conflicts will also be much higher. [15].

In this chapter, we discuss the issues of concurrency control and transaction scheduling for MDRTDBS which is supporting soft or firm real-time transactions. An important consideration in the design of the strategies for transaction processing and concurrency control is how to reduce the impact of the unreliable network on the performance of the system.

2. SYSTEM AND TRANSACTION MODEL

2.1 SYSTEM ARCHITECTURE

Basically, a MDRTDBS consists of four main components: mobile clients (MCs), base stations, mobile network, and main terminal switching office (MTSO) [9], as shown in Figure 18.1. The mobile network is assumed to be a low bandwidth cellular network such as the GSM network. The entire service area is divided into a number of connected cell sites. Within each cell site, there is a base station, which is augmented with a wireless interface to communicate with the MCs within its cell site.

The base stations at different cell sites are connected to the MTSO by a point-to-point wired network. Thus, the communications between the base stations and the MTSO are much more efficient and reliable than the communications between the base stations and the mobile clients. The MTSO is responsible for active call information maintenance, performance of handoff procedure, channel allocation and message routing. Attached to each base station is a real-time database system containing a local database which may be accessed by transactions issued by the MCs within the cell or from other cells via the MTSO.

An MC may move around within the same cell or cross the cell border into another cell. Periodically, it sends a location signal to its base station through an up-link channel. When an MC is crossing the cell border, the MTSO will perform a handoff procedure to assign the MC to the new base station.

The transactions generated from MCs may need to access databases located at several base stations. The MC first issues a call request to the base station

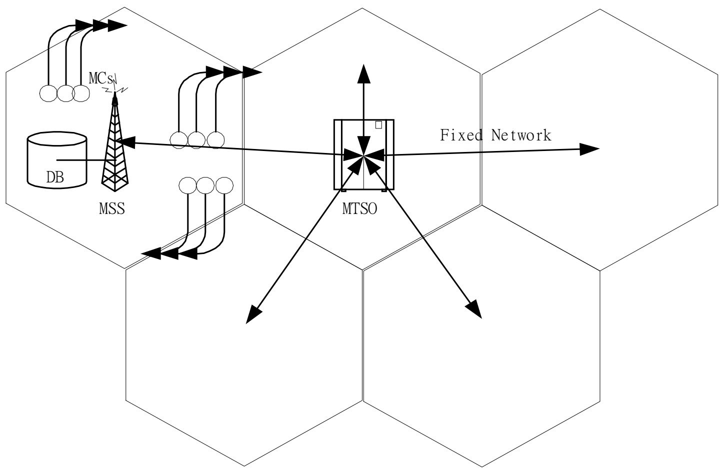

Figure 18.1 System Architecture of a MDRTDBS

of its current cell. A channel is granted to the MC after the completion of a setup procedure. Since the number of channels between a base station and its MCs is limited, it is possible that the channel request may be refused due to unavailability of free channels. Due to channel contention and slow (and unreliable) communication, the time required to establish a channel is unpredictable. Once a channel has been established, the transaction will be sent out through the RF transmitter from the MC to the base station. Due to noise and interference, the signal, which carries data, may be corrupted while it is being transmitted. In this case, the data will be re-transmitted. If the transmission of signals is corrupted consistently after several times, a disconnection may have occurred. For that case, the transaction may need to wait until a new channel is granted before it can proceed. Because of high error rate and non-stability of signal transmission, the effective data transmission rate is also unpredictable.

2.2 DATABASE AND TRANSACTION MODELS

The entire database is partitioned into local databases and distributed at different base stations. The databases consist of two types of data items: *temporal* and *non-temporal* [14, 18]. Transactions from the MCs are assumed to be simple flat transactions. Each transaction consists of a set of read and write operations. They are associated with a soft or firm deadline. In between the operations of a transaction, control statements may be defined to control the logic flow of the transaction. Examples of such application systems are jockey-club betting systems, Internet programmed stock trading systems, traffic navigation and information systems, etc. Each transaction is given a criticality. The priority of a transaction is then derived based on its deadline and criticality. It is assumed that the EDF algorithm is used for scheduling the transactions in using the CPU. Operations may access data objects residing at different base stations. Thus, a transaction may have several processes, called *transaction processes*, at different base stations for its execution. When an operation of a transaction accesses a data object residing at another base station, the operation will be routed to the base station via the MTSO and a new transaction process will be created if there is no process at that site for the transaction. When all the operations of a transaction have been processed, a commit protocol will be performed to ensure the failure atomicity of the transaction.

3. CONCURRENCY CONTROL PROTOCOL FOR MDRTDBS

Concurrency control is important to ensure consistency of database and data items observed by transactions. Due to the unique properties of a MDRTDBS, in the design of concurrency control protocols for MDRTDBS, there are two important considerations:

1. how to minimize the cost and overheads for resolving data conflicts; and
2. how to minimize the impact of mobile network on the performance of the protocol.

In the last decade, many efficient real-time concurrency control protocols have been proposed. Usually, they use transaction restart or priority inheritance to resolve the problem of priority inversion. However, restarting a transaction is highly expensive in a mobile environment due to the slow and unreliable mobile network. Although priority inheritance [4] is effective in managing the priority inversion problem in single-site RTDBS, a direct application of it may not be effective in a MDRTDBS as it may take a long time before the priority of a transaction is "inherited". Furthermore, deadlock is possible when priority inheritance is used. Deadlock is highly undesirable to real-time systems, especially in a distributed environment. The detection and resolution of a deadlock in a distributed environment may also consume a lot of resources. Thus, a deadlock-free protocol is highly desirable. Although the real-time optimistic concurrency control protocols have been shown to give a good performance in single-site RTDBS [3], they may not be suitable to MDRTDBS. The validation test required in optimistic concurrency control protocols can be very complex in a distributed environment [10] especially in mobile networks.

In the following, we introduce a real-time concurrency control protocol, called Distributed Higher Priority Two Phase Locking [1], for MDRTDBS. Although the basic methods used to resolve data conflicts are still based on transaction restart and priority inheritance, the protocol consists of two main features which are specially design for mobile environment. Firstly, it adopts a state conscious method to resolve data conflicts, e.g., the method used to resolve a data conflict depends on the status of conflicting transactions. Secondly, methods are included to reduce the cost of transaction restarts and prevent the possibility of deadlock when priority inheritance is used.

3.1 THE PROTOCOL

The definition of the DHP-2PL is as follows, where T_r and T_h, are the lock-requesting transaction and the lock-holding transaction, respectively:

```
LockConflcit (τr, τh)
    Begin
        If Priority(τr) > Priority(τh)
            If τh is not committing
                If τh is a local transaction
                    Restart τh locally
                Else
                    Restart τh globally
                Endif
            Else
                Block τr until τh releases the lock
                Priority(τh):= Priority(τr) + fixed priority level
            Endif
        Else
            Block τr until τh releases the lock
        Endif
    End
```

A transaction is local if it only accesses data objects resided at one base station. Otherwise, it is a global transaction. Restarting a local transaction is simply done by restarting the transaction process at the conflicting base station. To restart a global transaction, restart messages are sent to the base stations where some operations of the global transactions are executing or have executed. Global restart takes a much longer time and requires a much higher overhead. Thus, the number of transaction restarts, especially a global one, should be minimized and only non-committing transactions will be restarted. Committing transactions are allowed to hold locks until they have finished the commit procedure even though higher-priority transactions are requesting the locks. Although this approach may create the priority inversion problem, the blocking time of the higher-priority transaction will not be long if the committing transaction is assigned a sufficiently high priority by using priority inheritance. The priority of the committing transaction will be raised up by two factors. Firstly, its priority will be at least as high as the highest priority of all of its blocked transactions, and, secondly, a fixed priority level should be added to its priority to make it even higher than all other executing transactions. The purpose is to finish the committing transaction as soon as possible. The time required to raise the priority of a transaction should be short, as the conflicting processes are located at the same site. No deadlock is possible for the priority raising of any committing transaction. It is because the committing transaction will not be blocked by any other executing transactions, as the committing transaction will not make any lock request during its commitment.

A common characteristic of mobile network is that disconnection between a mobile client and its base station is frequent. In processing a transaction, the control of a transaction may flow between its processes at the base stations and the process at its originating mobile client. In case a disconnection occurs while a transaction is locating at the mobile client (the control flow of the transaction is at the mobile client), the impact of the disconnection on the system performance can be very serious due to chain of blocking. It does not only greatly increase the deadline missing probability of the disconnected transaction, other transactions, which are directly or transitively blocked by the disconnected transaction, will also be affected. The result may be *fruitless blocking* which is a situation where a blocked transaction is finally aborted due to deadline missing. In the protocol, the problem of fruitless blocking may occur when a lower-priority transaction is blocked by a higher-priority transaction, which is a disconnected transaction. To minimize the impact of disconnection and the probability of fruitless blocking due to disconnection, we may use a *cautious waiting* scheme in which a higher-priority transaction is restarted by a lower-priority transaction due to data conflict if the higher-priority transaction is suspected to be a disconnected transaction.

Let each executing transaction process be associated with a location indicator. When a transaction is at the mobile client or it is waiting to move back to the mobile client, the location indicator of its processes will be set as "mobile client". Otherwise, it is set as "base station". When the location indicator of a transaction process is "mobile client", the transaction is vulnerable to disconnection. The following summarizes how the cautious waiting scheme is incorporated into the DHP-2PL:

```
If (the priority of the lock-requester > the priority of the lock-holder)
   and (the lock-holder is not committing)
      Restart the lock-holder (globally or locally, depending on
      the type of the transaction);
Else
      If the location indicator of the lock-holder is "mobile client"
         If the time already spent at the client side > threshold
            Ping the mobile client where the lock-holder is residing;
            /* The base station sends a message to the mobile client to
                test if the mobile client is disconnected or not. */
            If no response from the mobile client
                Restart lock-holder;
            Else
            Block the lock-requester;
            /* repeat the checking after another threshold */
            Endif
         Else
            Block the lock-requester;
            /* the checking will be performed again when the time
                already spent at the client side is greater
                than the threshold value*/
         Endif
      Else
         Block the lock-requester;
      Endif
   Endif
```

The *threshold* is a tuning parameter. It is a function of the average performance of the mobile network under normal situations. If a transaction has been at the mobile client for a long time, e.g., greater than the threshold value, and its base station cannot communicate with the mobile client currently, disconnection is assumed. The lock-holding transaction will be restarted even though its priority is higher than the priority of the lock-requesting transaction. Although restarting the lock-holding transaction will make it have a high probability of missing deadline, the restart will not significantly affect the system performance, as the lock-holding transaction is likely to miss its deadline due to disconnection. On the other hand, restarting the lock-holding transaction can increase the chance of meeting the deadline of the lock-requesting transaction. Otherwise, it is highly possible that both of them will miss their deadlines. Note that simply making the assumption of disconnection based on pinging the mobile client may not be sufficient as mobile networks are subjected to different transient communication failures, which are much less harmful than disconnection. A transient communication failure will only last for a very short time and can usually be solved by data re-transmission.

4. STRATEGY FOR PROCESSING OF TRANSACTIONS IN MDRTDBS

Although the DHP-2PL can significantly reduce the cost for resolving data conflicts, the cost of transaction restart can still be substantial if conventional transaction processing strategies, e.g., query shipping or data shipping, are used to process the transactions. In this section, we discuss a new strategy, called *transaction shipping* to process the transactions from mobile clients. The goal of the transaction shipping approach is to further reduce the impact of a mobile network on the performance of the DHP-2PL and to further improve the system performance.

4.1 TRANSACTION SHIPPING APPROACH

Conventional methods designed for processing transactions in distributed systems supported by wired network, such as data shipping or query shipping, may not suitable to the systems designed on mobile network as the communication overhead can be very high. This is highly undesirable especially when it is combined with a restart based concurrency control protocol. To reduce the overhead, we may "*ship*" the entire transaction to the database server (base station) for processing instead of shipping every operation or data request to the database server. The new method is called *transaction shipping*. Although the idea of shipping the whole transaction is simple, there exist many practical problems when it is applied to a MDRTDBS such as how to identify the execution path and the required data objects of a transaction before its execution, and how to deal with the dynamic properties in transaction execution. In the transaction shipping approach, the problems are resolved by a *pre-analysis* scheme in which the characteristics of a transaction will be defined and estimated before its start of execution. The practicality of the pre-analysis comes from the observation that the *behavior of many real-time transactions is more predictable comparing to the transactions in conventional database systems.* To deal with the dynamic properties of the real-time transactions, a *data pre-*

fetching mechanism is included in the transaction shipping approach to reduce the cost of incorrect prediction in the pre-analysis.

4.1.1 Transaction Predictability. Transactions in a real-time database application can be classified into different types based on their pre-defined behavior and characteristics. Different transaction types will have different pre-defined behavior and critically [7]. For example, in a medical information system, periodic hard real-time transactions are used for monitoring the physical status of a critical patient using various sensor devices, such as blood pressure, heart beat rate, and body temperature. In some real-time database applications, e.g., programmed stock trading, although the transaction arrival pattern may be sporadic, their data requirements can be predicted with a high accuracy. For example, in a programmed stock trading, each investor may have a pre-defined investment plan, e.g. what their interested stocks are and how to make the trading analysis under different conditions. In a traffic navigation system, the physical connections of the roads are pre-defined. When searching the best path to a destination from the current position based on the current road conditions, the set of roads to be search is also pre-defined.

4.1.2 Pre-analysis Phase. In the transaction shipping approach, the execution of a transaction is divided into two phases: the *pre-analysis phase* and the *execution phase*. Figure 18.2 shows the transaction architecture when it is processed under the transaction shipping approach. Once a transaction is initiated at a mobile client, a coordinator process, called *master coordinator*, will be created at the mobile client. Before shipping a transaction to the base station, the system will perform a pre-analysis on the transaction to derive its characteristics, e.g., what the operations of the transaction are and what the execution path of the transaction is.

The pre-analysis of a transaction consists of two stages. In the first stage, the *set of operations* in the transaction will be identified. It is usually not difficult to identify the operations in a transaction. For example, if SQL statements are used to access the database, the SELECT statements are read operations and the INSERT statements are write operations. At this stage, it may not be necessary to identify the set of data objects required by the operations. Actually, it may not be easily done at the mobile client as it only contains limited information about the database system and the location of the data objects in the system. The required data objects of an operation will be determined while the transaction is executed at the base stations.

In the second phase, the *execution path* of the transaction, e.g., the precedence relationships of the operations, will be determined. However, for some transactions, the whole execution path may only be determined until the data

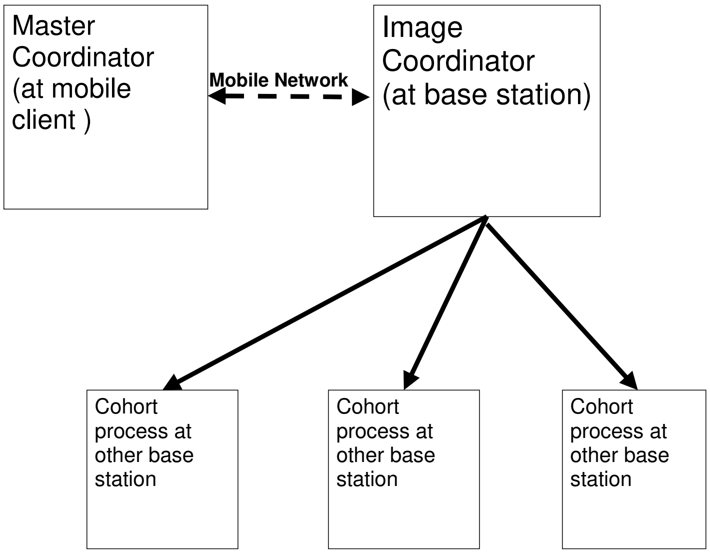

Figure 18.2 Transaction Architecture under the Transaction Shipping Approach

objects required by the transactions have been identified. For example, some conditional statements are based on the values of the data objects. For such transactions, the pre-analysis may identify the transaction type first and then make the prediction based on the pre-defined characteristics of that transaction type.

After the completion of the pre-analysis, a signature of the transaction will be created. A signature transaction S_i for transaction T_i consists of a 4-tuple:

$$S_i = (O_i, D_i, C_i, <_i)$$

where D_i, C_i, and $<_i$ are the deadline of T_i, a subset of the operations in T_i, and the partial order relationship among operations in O_i, respectively. Note that $<_i$ defines the precedence relationship among the operations. If $Op_j <_i Op_k$, then Op_k can start its execution only after the completion of Op_j, where Op_j and Op_k are operations of T_i.

4.1.3 Execution Phase. The signature transaction is forwarded to the base station of the MC through the mobile network. Once the server at the base station receives the transaction signature, it will create a process, called *image* *coordinator*, for the transaction. The image coordinator will take over the job from the master coordinator to process the transaction. Other transaction processes for the executions of the transaction will be created at other base stations if the operations of the transaction require to access the data objects located at that base stations.

The benefit of defining an image coordinator at the base station is that the connection between base stations is much better than the connection between the MC's and their base stations. Thus, putting the image coordinators at the base stations can facilitate the management of the transactions and improve the performance of the atomic commitment protocol. Whenever a transaction has to be restarted, all its processes (excluding the master coordinator and the image coordinator) will be destroyed after the completion of undo operations. The image coordinator is responsible for restarting the transaction from its beginning if its deadline has not been missed.

4.1.4 Data Pre-fetching Mechanism. Although the transactions in real-time database applications are more well-defined, they may still have some dynamic properties. As a result, the pre-analysis of a transaction may need to be re-done while it is executing. For that case, it has to go back to its originating MC. Before a transaction goes back to the mobile client, the system may pre-fetch the data objects possibly needed by the next operation of the transaction, and then ship the data objects with the transaction back to the mobile client. The purpose of the *data pre-fetching mechanism* is to process the next operation of the transaction at the mobile client so that the total number of communications between the mobile client and the base station can be reduced.

The assumption of the data pre-fetching mechanism is that data requirements of the next operation can be predicted with a high accuracy. Again, the validity of the assumption is based on the well-defined behavior of real-time transactions. Suppose the execution path of a transaction T_1 is originally predicted to be Path 1 (operations: Op_1, Op_2 and then Op_3) in the first pre-analysis as shown in Figure 18.3. After the completion of operation Op_2, it finds out that the original prediction is incorrect, and T_1 has to go back to the mobile client to "re-define" the execution path which is either Path 2 or Path 3 according to the pre-defined behavior of the transaction. Before, T_1 is shipped back to the mobile client, the system pre-fetches the data objects required by Op_4 and Op_5. Later at the mobile client, T_1 can process the next operation Op_4 or Op_5. After that, T_1 is sent to the base station to commit or continue its following operations. Note that under the data pre-fetching mechanism, T_1 has to set a lock on the pre-fetched data objects. The purpose to is to ensure the serializability of transaction execution.

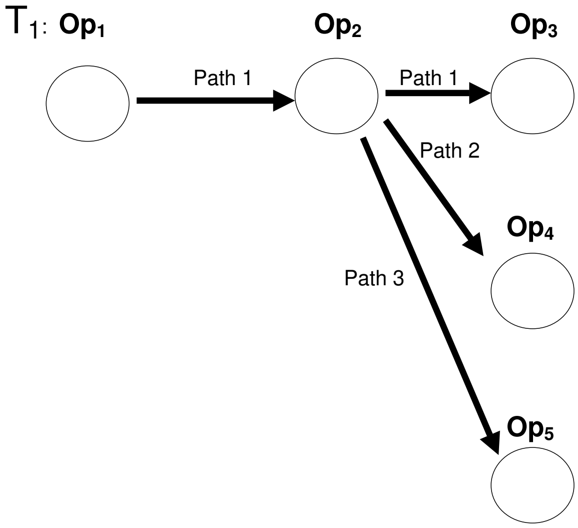

Figure 18.3 Data Pre-fetching Mechanism

Note that the transaction shipping approach does not require the transmission of a large number of data objects to the clients as required in the data shipping approach. This is an attractive feature to MDRTDBS as the communication bandwidth is low. Although some data objects are required to be shipped with a transaction in the data pre-fetching mechanism, the data volume should be much smaller than that required in the data shipping approach as the data pre-fetching mechanism is required only when the predicted execution path is incorrect. Furthermore, only those data objects, which are required by the next operation, are shipped.

It is obvious that the performance gained from the pre-analysis depends on its accuracy. In the best case, one single communication between the MC and the base station is required for transaction execution. In the worst case, the number of communications between the mobile client and the base station for each transaction execution is equal to the number of operations in the transaction. If the system can make a good estimation in the pre-analysis, a lot of communications can be saved.

5. RELATED WORK

Although MDRTDBS are important, not much work has been done until recent years. The impact of mobile network on the performance of the Higher Priority Two Phase locking is studied in [11]. In [8], the various issues to process real-time transactions in a MDRTDBS are examined. Two execution strategies, called Execution Site is a Fixed Host (ESFH) and Execution Site is a Mobile Host (ESMH), are proposed for mobile transactions. The tradeoff of the two strategies is on the processing power of the mobile hosts. The ESFH is similar to the transaction shipping approach except that it uses the mobile host to handle the user requests instead of using an pre-analysis approach as used in the transaction shipping approach.

6. SUMMARY AND FUTURE WORK

The design of mobile distributed real-time database systems (MDRTDBS) is receiving growing interests. Due to the poor quality of services provided by a mobile network, it is not easy to meet the deadlines of transactions in a MDRTDBS. In this chapter, we define a detailed model for MDRTDBS, in which the mobility of the mobile clients and characteristics of the mobile network are modeled explicitly. We introduce a distributed real-time locking protocol, called Distributed High Priority Two Phase Locking (DHP-2PL) for MDRTDBS, where the characteristics of the mobile network and the status of the conflicting transactions are considered in resolving the conflicts in data accesses. Then, we introduce a new strategy, called transaction shipping with a data pre-fetching scheme, to process the transactions to improve the system performance and to reduce the impact of the mobile network on the system performance.

Due to the advances in mobile communication technology and the need to management real-time information over a mobile network, it is a trend to "mobilize" the database applications and include various real-time features into the mobile computing systems. Thus, the importance of MDRTDBS is growing. We can see that the technologies from the research in MDRTDBS will be useful in many related research areas such data broadcast of real-time data, tracking of moving objects and the support of continuous queries.

Acknowledgements The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1078/00E].

References

[1] R.J. Abbott and Hector Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *ACM Transactions on Database Systems*, vol. 17, no. 3, pp. 513-560, Sep. 1992.

[2] A. Datta, A. Celik, J. Kim and D.E. VanderMeer, "Adaptive Broadcast Protocol to Support Power Conservant Retrieval by Mobile Users", *in Proceedings of 13th International Conference on Data Engineering*, 1997.

[3] J.R. Haritsa, M. Livny, M.J. Carey, "On Being Optimistic about Real-Time Constraints", in *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, 1990.

[4] J. Huang, J. Stankovic and K. Ramamritham, "Priority Inheritance in Soft Real-Time Databases", *Journal of Real-Time Systems*, vol. 4, no. 3, pp. 243-268, 1992.

[5] T. Imielinski and B. R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management," *Communications of the ACM*, vol. 37, no. 10, Oct. 1994.

[6] T. Imielinski and S. Viswanathan, "Adaptive Wireless Information Systems," *Proceedings of Special Interest Group on Database Systems (SIGDBS)*, pp. 19-41, Japan, 1994.

[7] Y.K. Kim, S.H. Son, "Supporting Predictability in Real-Time Database Systems", *Proceedings of Real-Time Technology and Applications Symposium*, Brookine, Massachusetts, June 1996, pp. 38-48.

[8] E. Kayan and O. Ulusoy, "An Evaluation of Real-Time Transaction Management Issues in Mobile Database Systems", *The Computer Journal*, vol.42, no.6, 1999.

[9] William C. Y. Lee, *Mobile Cellular Telecommunications Systems*, New York, McGraw-Hill, 1989.

[10] K W Lam, K Y Lam and S L Hung, "Distributed Real-time Optimistic Concurrency Control Protocol", in *Proceedings of International Workshop on Parallel and Distributed Real-time Systems*, Hawaii, April 14-15, 1996, pp. 122-125.

[11] V.C.S. Lee, K Y Lam and Wai-Hung Tsang, "Transaction Processing in Wireless Distributed Real-time Databases", in *Proceedings of 10th Euromicro Workshop on Real Time Systems*, Berlin, 1998.

[12] E. Pitoura and B. Bhargava, "Dealing with Mobility: Issues and Research Challenges", Technical Report, Purdue University, 1993.

[13] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 4, pp. 513-532, 1995.

[14] K. Ramamritham, "Real-time Databases", *International Journal of Distributed and Parallel Databases*, vol. 1, no. 2, pp. 199-226, 1993.

[15] O. Ulusoy, "Real-Time Data Management for Mobile Computing", in *Proceedings of International Workshop on Issues and Applications of Database Technology (IADT'98)*, Berlin, Germany, 1998.

[16] L. Sha, R. Rajkumar, S.H. Son, and C.H. Chang, "A Real-time Locking Protocol", *IEEE Transactions on Computers*, vol. 40, no. 7, pp. 793-800, 1991.

[17] P. Xuan, O. Gonzalez, J. Fernandez, and K. Ramamritham, "Broadcast on Demand: Efficient and Timely Dissemination of Data in Mobile Environments", in *Proceedings of 3rd IEEE Real-Time Technology Application Symposium*, 1997.

[18] M. Xiong, K. Ramamritham, R. Sivasankaran, J.A. Stankovic, and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics", *Proceedings of IEEE Real-Time Systems Symposium*, pp. 240-251, 1996.

PART VI

PROTOTYPES AND FUTURE DIRECTIONS