TCOT – A Timeout-based Mobile Transaction Commitment Protocol

Vijay Kumar*, Nitin Prabhu
SICE, Computer Networking
University of Missouri-Kansas City
Kansas City, MO 64110
kumarv(npp21c)@umkc.edu

Maggie Dunham†, Ayşe Yasemin Seydim
Dept of Comp. Sc. and Eng.
Southern Methodist University
Dallas, Texas 75275-0122
mhd(yasemin)@engr.smu.edu

Abstract: We present a transaction commit protocol "Transaction Commit On Timeout (TCOT)" based on a "timeout" approach for Mobile Database Systems (MDS), which can be universally used to reach a final transaction termination decision (e.g., commit, abort, etc.) in any message oriented system. Particularly suited for a wireless environment, a timeout mechanism is the only way to minimize the impact of the slow and unreliable wireless link. We compare TCOT to a modified version of 2PC to show its superiority based on commit time.

Index Terms: Cell, Compensation, Execution fragment, Handoff, Mobile Database System, Mobile transaction, Timeout, Commit.

1 Introduction

The architecture of the *Mobile Database System (MDS)* we are investigating is based on *Personal Communication Systems (PCS)* where a number of *DBSs (Data Base Servers)* are assumed to exist in the fixed network. Components of the MDS include *Base Stations* (*BS*) or *Mobile Support Stations* (*MSS*), and a number of mobile computers (laptop, PDAs, etc.) referred to as *Mobile Hosts* (*MH*) or *Mobile Units* (*MU*), which are connected to the wired network through *BSs* via wireless channels [17, 9]. *MUs* are mobile processors with some database functionality, which includes cache management, transaction processing and roll-back, etc. Some of the common MU limitations are: (a) a *MU* may cease to communicate with its *BS* for a variety of reasons, (b) it may run out of its limited battery power, (c) it may run out of disk space, (d) it may be affected by airport security, physical abuse, or accident, (e) limited wireless channels for communication, and (f) unpredictable *handoff*s.

Transaction processing in a *Mobile Database System (MDS)* will be quite diverse, probably with many different transaction models and processing modes. Some may follow the

*This research is supported by the National Science Foundation under Grant No. IIS 9979453.
†This research is supported by the National Science Foundation under Grant No. IIS 9979458.

traditional ACID requirements while others may not. Furthermore, legacy commit protocols such as *2PC (two phase commit), 3PC (three phase commit)* [2], etc., could be molded to work with MDS, however, they may not perform satisfactorily mainly because of the limited resources, especially wireless channel availability.

In this paper we address transaction commitment in MDS systems. Our objective is to develop an efficient commit protocol for this highly resource-constraint dynamic environment which (a) should use minimum number of wireless messages and (b) *MU* and *DBSs* involved in transaction processing should have independent decision making capability and the protocol should be *non blocking* [2].

There have been many different models proposed for *Mobile Transactions (MT)* [6, 14, 4, 19, 20]. The proposed transaction commit protocol, will work under these. We briefly indicate the salient features assumed to exist to support our proposed commit protocol. An *MT* T_i is defined as T_i=$\{$ e_{i1}, e_{i2}, ..., $e_{in}\}$, where each e_{ij} is an "execution fragment" [11, 12] and represents a portion of the total transaction. Each T_i is requested at an *MU* and each e_{ij} is executed at either that *MU* or a set of *DBS* or at both places. We refer to the *MU* where a T_i originated or initiated as *H-MU* (Home *MU*) and the *BS* where *H-MU* initially registered as *H-BS* (Home *BS*). The objective of our commit protocol is to commit T_i in this environment. We also deal with fragment compensation if T_i can not be committed as a whole. Henceforth, we drop the first subscript of e_{ij} to obtain e_j.

Like conventional distributed database systems, in MDS also a *coordinator (CO)* is required to manage the commit of T_i. The *BS* or a node in the fixed network is the most suitable component for CO. The *MU* is not a good choice because it has (a) no direct communication with other processing nodes, especially with *DBS*s (b) limited storage and reliability, (c) limited power supply, (d) unpredictable *handoffs*, and (e) limited availability. However, in the case where T_i is entirely processed at the *H-MU*, it can act as a CO. We assume that each T_i has its own CO. The *commit set* is defined as the set of *DBS* and the *H-MU* that take part in the processing and commit of a T_i.

In Section 2 we present TCOT commit protocol, Section 3 examines the timeout values, and Section 4 present the result of performance study. Finally, Section 5 concludes the paper.

2 TCOT: Transaction Commit on Timeout Protocol

In TCOT, if the CO node does not receive a failure message from a node in the commit set within a predefined timeout period, then T_i commits. It is well known that finding the most appropriate value of a timeout is not always easy because it depends on a number of system variables, which could be difficult to quantify [2, 1]. However, it is usually possible to define a value for timeout, which performs well in all cases. We define two types of timeout: Execution Timeout and Update Shipping Timeout.

Execution Timeout (E_t) defines an upper bound timeout value within which a node of a commit set completes the execution (not commit) of its e_i. The value of E_t may be node specific. It may depend on the size of e_i and the characteristics of the processing unit. We identify *H-MU's* timeout by $E_t(MU)$ and *DBS* timeout by $E_t(DBS)$. The relationship between these two timeouts is $E_t(MU) = E_t(DBS) \pm\Delta$. The Δ accounts for the characteristics such as poor resources, disconnected state, availability of wireless channel, etc., compared to *DBS*. Furthermore, the value of a timeout for an e_i depends on its *MU*, thus, $E_t(MU_i)$ may not be equal to $E_t(MU_j)$, $(i \neq j)$. It is possible that a *MU* may take less time than its E_t to execute its e_i. We also do not rule out the possibility that in some cases $E_t(DBS)$ may be larger than $E_t(H\text{-}MU)$. E_t typically should be just long enough to allow a fragment to successfully finish its entire execution in a normal environment (i.e., no failure of any kind, no message delay, etc.)

Shipping timeout (S_t) defines the upper bound of the data shipping time from *H-MU* to *DBS*. Thus, at the end of E_t the CO expects the updates to be shipped to the *DBS* and logged there within S_t. We compute S_t as *Time to compose updates* (U_t) + *Time for the updates to reach CO* (S_h).

In the remainder of this section we examine how TCOT works. We first explore its execution assuming no failure and then examine failure processing. The following steps, and associated Figure 1, explain the working of TCOT in terms of the activities of *H-MU*, CO, and *DBSs*. We first consider no failure of any kind and with no fragment *compensation*.

- *Activities of H-MU*:
  - A T_i originates at *H-MU*. The *H-BS* is identified as the CO. *H-MU* extracts its e_i from T_i, computes its E_t, and sends $T_i - e_i$ to the CO along with the E_t of e_i. *H-MU* begins processing of e_i.

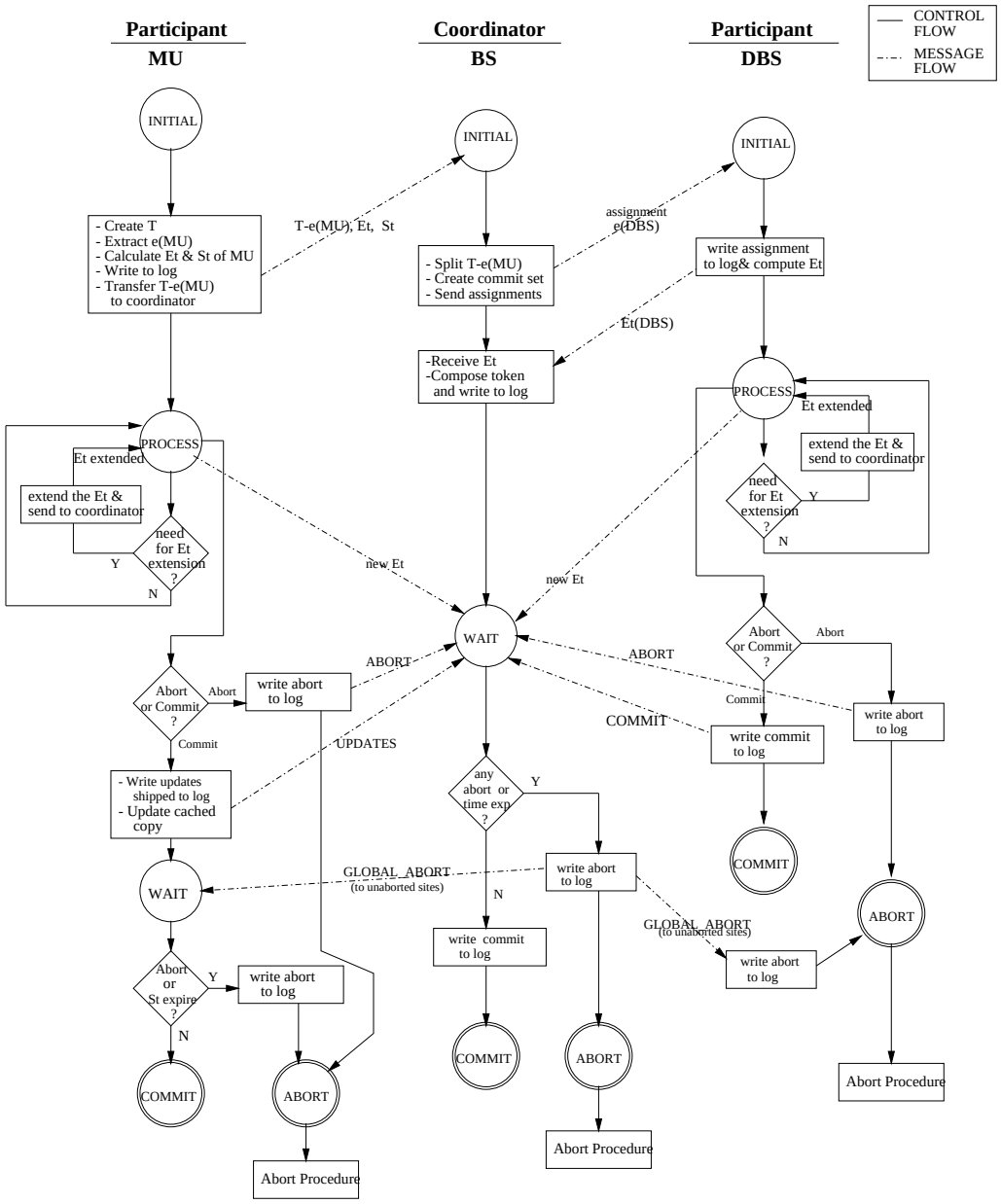

Figure 1: Execution of an MT in TCOT leading to commit.

– While processing e_i, *H-MU* updates its cache copy of the database, composes update shipment and appends it to the log. During processing if it is determined that e_i will execute longer than E_t, then *H-MU* extends its value and sends it to CO. If the local e_i aborts for any reason, then *H-MU* sends an Abort message to CO before E_t expires.
– After execution of e_i *H-MU* sends log of updates to the CO. The updates must reach to CO before S_t expires. It could be possible that updates may reach CO much earlier in which case it may decide commit sooner. In the case of read-only

e_i *H-MU* sends a commit message to CO[1].

- Once the updates are dispatched to CO, *H-MU* declares commit of e_i. If *H-MU* fails to send updates to CO within S_t and it did not extend E_t, then the CO aborts e_i.

- *Activities of CO*:
  - Upon receipt of $T_i - e_i$ from *H-MU*, the CO creates a `token` for T_i, which contains one entry for each of its e_i, E_t, CO's identity and commit set. In the case of CO change, a token is used to inform the new CO the status of e_i and commit set members. The CO splits $T_i - e_i$ into e_j's ($i \neq j$) and sends them to the set of relevant *DBSs*.
  - If a new E_t (extension) is received either from *H-MU* or from a *DBS*, then the CO updates the token.
  - CO logs the updates from *H-MU*. If the CO has *H-MU*'s shipment before S_t expires and commit messages from other *DBSs* of the commit set, then the CO commits T_i. At this time the updates from the *H-MU* are sent to the *DBSs* for update to the primary copy of the databases. Note that no further message is sent to any member of the commit set of T_i.
  - If CO does not receive updates from *H-MU* in time or does not receive commit message from any of *DBSs* of the commit set, then it aborts T_i and sends a global abort message to those members of the commit set who committed their e_i's.

- *Activities of DBS*:
  - Each *DBS*, upon receiving its e_i, computes E_t and sends it to the CO, begins processing its e_i and updates its own database. If it is determined that the e_i will execute longer than E_t, then this value is extended and the new value is sent to the CO.
  - At the end of e_j it sends a "commit message" to the CO. If *DBS* cannot complete the execution of its e_j for any reason and did not extend E_t, then it sends an Abort message to the CO.

Ti commits when CO has received all commit messages and updates and for members their fragments commit when they have sent commit messages and updates. If a T_i fails to

[1]Note that this is not an extra message, it just replaces shipping update message.

commit, then it may initiate cascade rollback if the underlying concurrency control[2] unlocks e_i's data items before global commit. This can be resolved if T_i's are run serially or if the concurrency control unlocks data items after global commit. Both these schemes will affect TCOT's performance. In any case it cannot be solved by any commit protocol alone and there is no way to completely avoid this problem when the independent functioning of the commit set members is assumed.

We have used a compensation scheme to manage this situation. Although the process of compensation is not a commit issue rather it is a part of recovery [8], we briefly explain its processing. If a T_i is aborted, any independently commited e_i fragments must be compensated. We refer to this as ("fragment compensation"). Each node in the commit set is responsible for the compensation of fragments which execute there. At the end of compensation, the node informs the CO. We realize that not all cases can be compensated [8] and again it is the job of the scheduler to manage this aspect. We argue that compensating approach should not affect TCOT's performance because time of compensation is not a part of CT (commit time). We discuss the effect of aborts in the performance section.

Since no global commit message is sent to any member of the commit set, it is important to know when a T_i actually commits. The absence of an abort message after E_t expires indicates a global commit. Thus the commit time is the time indicated by E_t. A premature abort is indicated by an abort message.

In TCOT although messages flow from members to CO but the latter actually neither wait for such messages nor for any information from the members of the commit set of T_i. All these happen in one phase. In 2PC the CO waits (expects) for messages from participants to make any decision. If the wait is over unsuccessfully, then it aborts the transaction. In TCOT absence of a message is enough to make a decision thus no additional phase is necessary.

Updates from *H-MU* and commit message from *DBS*s must be sent to the right CO if it changes in a handoff. The change in CO is notified using a *token* and commit proceeds as follows. *H-MU* moves to a different cell and it registers with the new *BS*. If CO changes, then *H-MU* sends the identity of the last CO to the next CO (new *BS*) during registration. The new CO then receives the `token` from the last CO. The new CO notifies other members

[2]We assume a strict 2-phase locking

of the commit set about the change of CO through wired channel.

A doze mode of *H-MU* will mainly affect its E_t. *H-MU* may not be aware of its movement, but it knows when it enters into the doze mode. Therefore, before going to doze mode *H-MU* can always request for extension to its E_t and if granted then e_i is not aborted otherwise a global abort is implied. Note that the CO sends abort message to commit set members only if S_t has expired or commit message did not arrive from at least one DBS.

Unlike E_t, S_t is not affected by T_i. Furthermore, the S_h component of S_t is more likely to change compared to U_t and it is possible to compute S_t so that the possibility of its extension is extremely small. We did not simulate S_t extension in our performance study.

A commit decision by a CO is said to be "correct" if the decision to commit is unanimous. We assume the contrary. Suppose the CO decides to commit T_i when at least one member of the commit set is undecided. This is possible only if the CO decides commit before the expiration of either S_t or the expiration of E_t and the absence of "commit message" from *DBS*s. This can not happen. Further, suppose that the *MU* failed and could not send updates to the CO within S_t or the "commit message" is not received by the CO. In this situation, the CO will abort T_i. Since our algorithm is based on timeout, it is not possible that at any stage the CO will enter into an infinite wait.

3 Analysis of Timeout Values

E_t is a crucial parameter and affects throughput and message cost. A small value of E_t may generate a large number of extension requests. While we can't really determine what E_t and S_t precisely should be, we can examine the impact of E_t values that are too short. In this case either the T_i is aborted because the time has expired or extensions are made to request the extension of E_t. As can be seen above, the number of wireless messages for a successful T_i is:

$$2 + N_{ext} \tag{1}$$

If the T_i does not abort because the time has expired, then Equation 1 indicates the total number of wireless messages. If the T_i does abort because of a timeout, we assume that the T_i will be executed again. Assuming that p_{ab} is the probability of abort caused by a timeout ($E_t + S_t$ is too low), we thus have the expected number of wireless messages when one abort occurs is:

$$(2 + N_{ext}) + p_{ab}(2 + N_{ext}) \tag{2}$$

In general, with n aborts this is then:

$$\sum_{i=0}^{n} p_{ab}^{i}(2 + N_{ext}) \tag{3}$$

In theory, the number of aborts could go on forever, so that we have an estimate of the number of wireless messages is finally:

$$\lim_{n\to\infty} \sum_{i=0}^{n} p_{ab}^{i}(2 + N_{ext}) \tag{4}$$

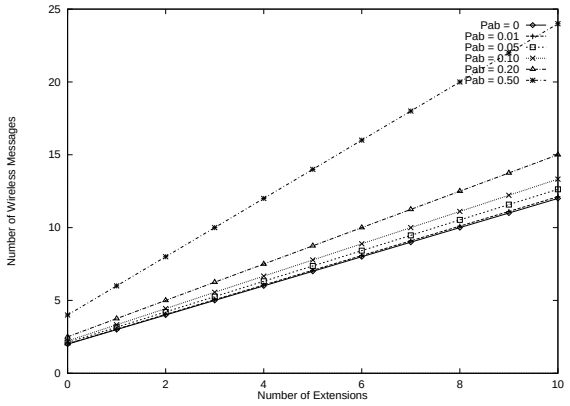

Figure 2: Number of Wireless Messages Based on Number of Extensions

Using Equation 4 we examine the impact of $E_t + S_t$ being too small on the number of wireless messages in Figure 2. The results for six different values of probability of abort are shown. The results for small values of p_{ab} are quite close. It is only as the p_{ab} gets values in the double digits that we see a significant increase in number of wireless messages. As the number of extensions increases, obviously the number of messages increases at a linear rate. While we can't predict exact values for p_{ab} and N_{ext}, we wouldn't expect either to be at the upper end of the values shown in the figure. Here probability of abort refers to the probability that a T_i will abort because the value of $E_t + S_t$ is too small. In this case, recall, the decision to abort is actually made by the CO because the update shipment has not been received in time. We, therefore, expect that the p_{ab} will be quite small. While the program may automatically request extensions as the time increases, we would also expect N_{ext} to be small.

To ensure small values for N_{ext} and p_{ab}, we propose that the value for E_t vary in the following manner:

- When a T_i aborts because the timeout values are too small, increase the timeout values.

Without more information about how short the time is, a rule of thumb could be to extend it by the $n \times E_t$ where $n = 1, 2, ...n$ for each subsequent rerun. Notice that this in effect reduces p_{ab} as the number of reruns increases.

- To ensure small values for N_{ext} whenever an extension is requested, the value for E_t is extended. We propose that the first extension be ext, the second be $2 \times ext$, and the n^{th} be $n \times ext$. This will reduce the number of extensions while trying to keep the total E_t value small.

4 Performance Comparison

In this section we report on simulation experiments conducted which compare the average *Commit Time (CT)* and the throughput of TCOT with a conventional 2PC. We compute CT as:

$$CT = S_t + \textit{time to compute } S_t + \textit{commit messages.}$$

CT starts when the first commit message or the update shipment (at the end of execution) is dispatched to the CO and it ends when CO declared $T_i's$ commit. We compare to 2PC as this is the most widely accepted commit protocol in use today. In actuality we use a modified 2PC (M2PC) [2]. M2PC uses one less message in the commit than traditional 2PC. We have implemented M2PC in such a way that its working is comparable to TCOT. The steps of the M2PC algorithm are as follows:

- T_i arrives at *H-MU*. If it cannot process the entire T_i, then it extracts its e_i and sends the rest (T_i-e_i) to the CO. The CO receives T_i-e_i, fragments it and distributes the e_i's among relevant *DBSs* along with the embedded "Vote Request" message to all members of the commit set.
- As soon as a member finishes its e_i, it sends a "Ready to Commit" message to the CO. The CO sends a "Commit" message to members of the commit set only if it receives "Ready to Commit" messages from all the members and declares commit of T_i. Only *H-MU* sends updates to the CO. If any member decides to abort its e_i, then it immediately sends an "Abort" message to the CO. The CO then sends "Abort" message to the rest of the members and aborts T_i.

In M2PC commit or abort information is determined as the T_i executes but in TCOT the CO has this information in the beginning of the execution of T_i, which helps to improve its efficiency. It is obvious that the total number of messages as well as the message rounds in M2PC will always be more than what is used in TCOT. Therefore, from the message viewpoint the performance of TCOT is superior. The time consuming activity in TCOT is the computation of E_t and S_t, which is not present in M2PC.

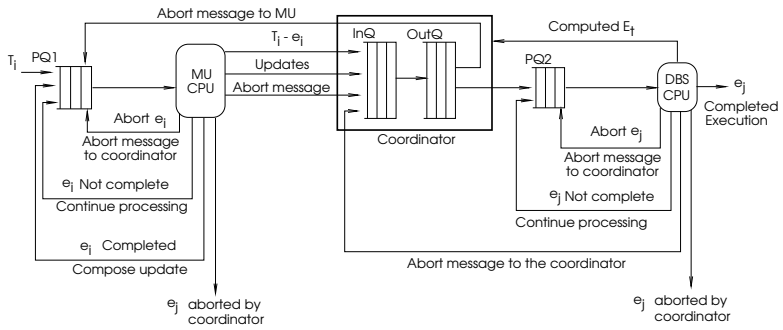

Figure 3: Simulation model of TCOT in MDS

Figure 3 shows the closed system simulation model (implemented using CSIM [5]) used for our experiments. We refer to the total number of active e_i's in MDS as the Multiprogramming Levels (MPL). The CO is shown as a set of queues. Every queue is processed in round-robin fashion. A T_i is always initiated from *MU* and arrives at the Pending Queue (PQ1). The *MU* extracts its e_i, computes its E_t, and sends T_i-e_i along with other information to InQ of the CO. An *MU* aborts its e_i on the request of the CO or unilaterally. The CO, if necessary, further fragments (T_i-e_i), and distributes the fragments to relevant *DBSs* through its OutQ. Each *DBS* estimates E_t for its fragments and sends it to the CO via PQ2.

An e_i at the *MU* may go through PQ1 a number of times during its execution. Every time an e_i is picked up from PQ1, its status is tested and if it has completed its execution, then it goes back to the end of PQ1 to compose the updates for shipment. If it is aborted, then it goes back to the front of PQ1 to compose and dispatch the abort message to the CO. At a *DBS* similar steps are followed, except no update is shipped to the CO.

We consulted a number of sources to identify a suitable value for message transfer time between *MU* and the CO and identified a data transfer rate of 10ms. [10, 3, 7, 15, 13]. Table 1 lists the values of fixed and variable parameters we used to drive the simulator. CPU times are calculated by estimated the number of instructions required for an operation.

Table 1: Simulation parameters

System Parameter	**Fixed value**
MU CPU speed	50 MIPS
DBS CPU speed	100 MIPS
No. of instructions to perform a Read from cache	1000
No. of instructions to perform a Write to cache	2000
No. of instructions to detect & resolve a conflict	2000
Time to perform an I/O	10 MS
Average Transaction Size	9 items
Maximum Number of *DBS*	4
Time to deliver a message over wired network	5 MS
Time to deliver message over wireless network	10MS
Variable parameters	**Value range**
Multiprogramming level (MPL)	1-1000
Probability of Update	0.0-1.0
Probability of Cache hit	0.0-1.0
Probability of Conflict	0.0-1.0
Probability of Hot Items	0.0-1.0
Transaction Size	2-10 Fragments

4.1 Performance Results and Discussion

We begin our experiment with the investigation into abort and handoff. Figure 4 and 5 examine the impact of abort and handoff on commit time. We assumed 10% abort and handoff for both protocols in all experiments[3]. We observe that there is a difference in the commit time, but the difference becomes less at higher MPL values. In all cases, the TCOT time is less than the corresponding M2PC time. Figure 5 shows the combined effect of abort and handoff. The trend is similar to Figure 4. The commit time of TCOT is less severely affected compared to M2PC. TCOT shows some resilience and its commit time changes significantly only after MPL = 60.

These results suggest that the effect of extra computation (computation of E_t, S_t, etc.) in TCOT does not affect its commit time significantly. Even in the stress situation generated by abort and handoff, TCOT manages to provide better commit time than M2PC, which is an optimized version of normal 2PC. It appears that message overhead is the main parameter affecting the commit time of M2PC more than it does to TCOT.

[3]We used this figure based on Sprint PCS statistics [15]

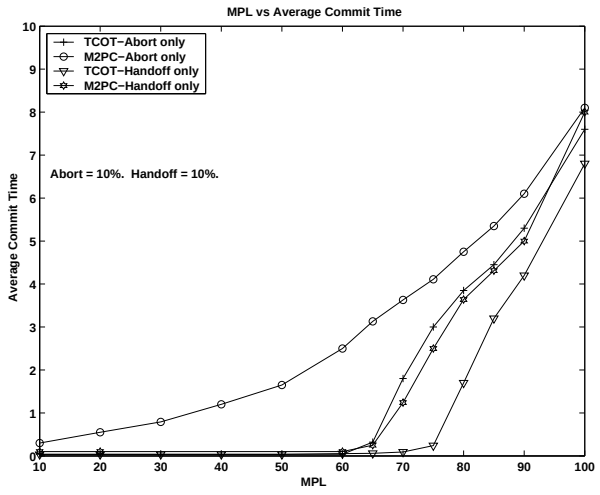

Figure 4: Effect of Abort and Handoff

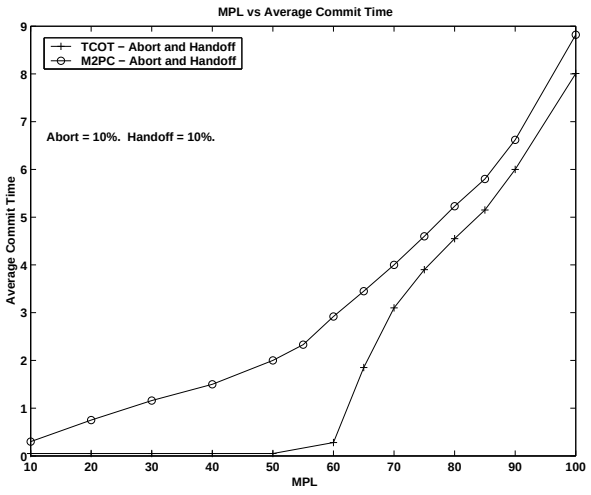

Figure 5: Effect of Abort and Handoff on CT

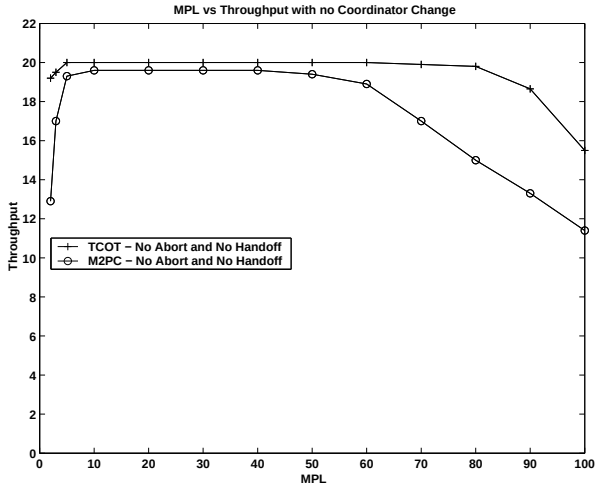

Figure 6: Throughputs of TCOT and M2PC.

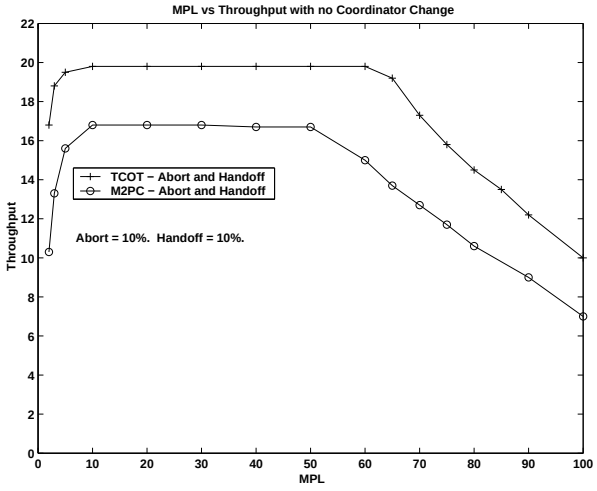

Figure 7: Effect of A and H on Throughput.

We next compared the throughput of TCOT and M2PC first with no coordinator change. In our earlier experiment (Figure 4 and 5), we observed that the effect of handoff is less than the effect of abort on commit time. This trend persisted in the throughput experiment as well. To show the performance shift from normal to stress situation we measured the throughput in these two situations. Figure 6 compares the throughput of TCOT and M2PC for the normal case (no aborts and no handoffs). We observe that TCOT offers higher throughput at all MPL values. The throughput of TCOT begins to decline after MPL = 80 but the throughput of M2PC begins to decline after MPL = 50. We measured the throughput beyond MPL = 100 but did not observe any significant change in the behavior except the throughput of M2PC became very low, which could be due to thrashing. The throughput of TCOT did not show such extreme behavior.

We also compared the throughput with abort only, with handoff only, and with abort and handoff combined together. We noticed that here also the effect of abort is more significant when compared to the effect of M2PC. We have included the figure of the combined case only. Figure 7 shows the performance with abort and handoff together. Note that this experiment includes random E_t extensions as well. We observe that the throughput of MDS under TCOT is significantly higher than M2PC and compared to the normal case (Figure 6) the throughput with TCOT does not show any significant change up to MPL = 60. This is not the case with M2PC where the throughput is noticeably less compared to the normal case. Note that in M2PC there is no request for E_t extension, thus no extra wireless message is required for processing extension requests. In the case of handoff, M2PC delays sending its "ready to commit message" which can be compared to the extension of E_t in TCOT.

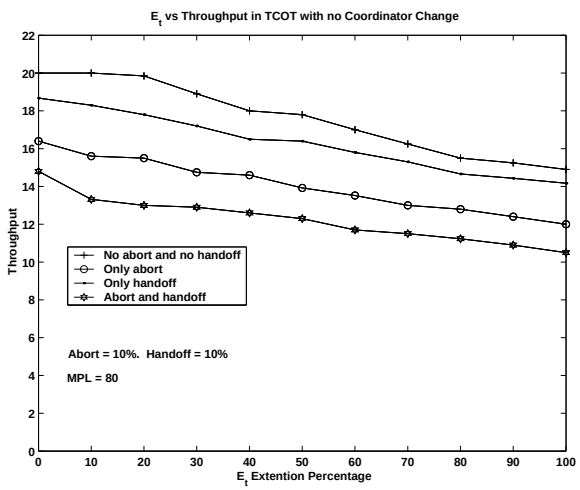

Figure 8: Effect of E_t on TCOT throughput

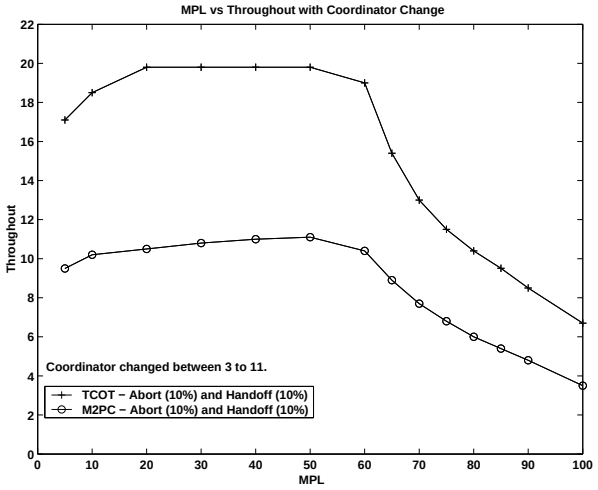

Figure 9: Throughputs with CO change

The effect of E_t change was studied analytically earlier. We verified the behavior with simulation. Figure 8 shows the behavior when the percentage of E_t was changed for MPL = 80. With each handoff, a fixed amount of delay was added to the execution time of a fragment. $E_t = 10$ means only 10% of the workload's requests for extension were granted. Some fragments were denied and some did not ask for any extension. We notice that the effect of E_t is not that severe on the throughput and the effect declines gradually. This could be further improved by reducing the amount of extension requests by carefully evaluating the initial value of E_t for each fragment. Note that our experiment included extension requests from *DBSs* too.

In the next experiment we randomly change the CO between 3 to 11. This is likely to be

most common scenario in MDS, especially with commuter and sales agents. Note that each cell (BS) has a CO and a change of, for example, 10 CO's, would cover 10 cells (about 10 to 15 miles area).

Figure 9 shows the throughput of TCOT and M2PC with coordinator change. Their throughput degrades, however, TCOT is still acceptable whereas M2PC degrades about 60%. Again after MPL = 60 value the throughput of both protocols degrades rapidly.

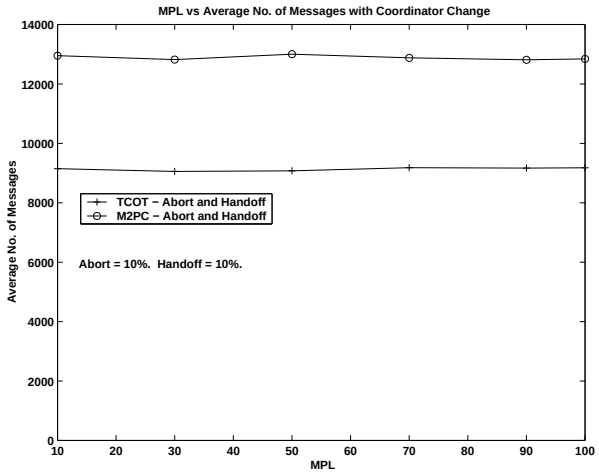

Figure 10: Average message utilization in TCOT and M2PC

We wanted to see the volume of wireless messages utilized by TCOT and M2PC in the entire process (execution and commit). In order to do this, we counted the average number of messages for each MPL in both protocols. We randomly changed the coordinators between 3 and 11. Figure 10 shows the number of messages utilized by each protocol. As expected, TCOT uses many fewer messages than M2PC. The number of messages do not increase with MPL because in each case the same number of transactions are executed and committed.

5 Conclusions

We presented a one phase commit protocol TCOT based on timeout approach. We extended the scope of the use of timeout by using it to identify the successful end of an activity. Thus, in our approach timeout not only enforces the termination condition but entire execution duration as well.

We compared its performance with a well-known modified 2PC (M2PC) commit protocol through simulation. We observed that in all the cases from normal to stress, TCOT

offered better performance in terms of commit time, throughput and significantly reduced the messaging cost.

References

[1] E. Bertino, E. Pegani, and G. P. Rossi. Fault Tolerance and Recovery in Mobile Computing Systems. In Recovery Mechanisms in Database Systems (Eds. V. Kumar and M. Hsu). Prentice-Hall, 1997.

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

[3] Imrich Chlamtac and Jasin Yi-Bing Lin "Wireless and Mobile Network Architecture", Wiley 2000.

[4] P. K. Chrysanthis, Transaction Processing in Mobile Computing Environment, In *IEEE Workshop on Adv. in Parallel and Dist. Sys.*, pages 77–82, October 1993.

[5] Mesquite Software Inc., *CSIM18 Simulation Engine (C++ Version).*

[6] M. H. Dunham, A. Helal, and S. Balakrishnan, A Mobile Transaction that captures both the data and movement behaviour, ACM/Baltzer Journal on Special Topics in Mobile Networks and Applications, 1997.

[7] Vijay K. Garg and Joseph E. Wilkes, "Wireless Personal Communication Systems", Prentice Hall PTR, 1996.

[8] Henry Korth, Eliezer Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *Proc. 16th VLDB Conf., Brisbane, Australia 1990.*

[9] M. Mouly, and M-B. Pautet, "The GSM System for Mobile Communications", Cell and Sys. (France), 1992.

[10] TIA/EIA Interim Standards-IS-95.

[11] V. Kumar and M. Dunham, Defining Location Data Dependency, Transaction Mobility and Commitment, TR 98-cse-1, Southern Methodist University, Feb. 98.

[12] V. Kumar, "A Timeout-based Mobile Transaction Commitment Protocol", *2000 ADBIS-DASFAA Symp. Adv. in Databases and Inf. Sys.*, In Cooperation with ACM SIGMOD-Moscow, September 5-8, 2000, Prague, Czech Republic.

[13] *http://wireless.networld.com/bandwidth.cfm*

[14] E. Pitoura and B. Bhargava, Maintaining consistency of data in mobile distributed environments, *Proceedings of 15th International Conference on Distributed Computing Systems.*, 1995.

[15] Personal Communication. Sprint Personal Communication Systems group.

[16] *http://www.storagereview.com/map/lm.cgi/seek?.*

[17] R. Kuruppillai, M. Dontamsetti, F. J. Cosentino, "Wireless PCS", McGraw-Hill, 1997.

[18] V. Kumar and S. H. Son, Database Recovery. Kluwer Academic Publishers, 1998.

[19] D. G. Walborn, and P. K. Chrysanthis, Supporting Semantics Based Transaction Processing in Mobile Database Applications. *Proc. 14th IEEE Symp. on Reliable Distributed Systems*, September 1995.

[20] L. H. Yeo, and A. Zaslavsky, submission of Transactions from Mobile Workstations in a Cooperative Multidatabase Processing Environment, *Proceedings of Distributed Computing Systems,* 1994, pp 372-379.