Single Phase Reliable Timeout Based Commit Protocol

Bharati Harsoor[1]
[1]Dept of CSE,
University College of Engg,
Osmania University, Hyderabad, India
bharsoor09@gmail.com

Dr. S.Ramachandram[2]
[2] Professor, Dept of CSE,
University College of Engg,
Osmania University, Hyderabad, India
schandram@gmail.com

Abstract - **The enormous progress in applications of distributed database systems necessitates formulation of an efficient atomic commitment protocol. The efficiency of these protocols is vital when higher transaction throughput is to be supported. The existing blocking commit protocols affect over the capacity of system resources, which worsens in distributed database system. This paper proposes the Non Blocking Single Phase Reliable Timeout Based Commit Protocol (SPRTBCP), an extension to the Modified Reliable Timeout Based Commit Protocol (MRTBCP), maintains the atomicity and supports off-line executions and disconnections during commitment; it decreases the cost of wireless communication by reducing it to a single phase commitment operation and does not maintain log agent. Hence it reduces message complexity and average commit time. It also supports disconnections and handoff in mobile environment.**

Keywords: Mobile Transactions, Disconnections, Handoff, Log Recovery, Atomic commitment protocols.

I. INTRODUCTION

The usage of portable devices equipped with wireless networks, is constantly increasing due to rapid progress in wireless technologies. These mobile devices also interact with fixed devices in realizing traditional applications like mobile commerce (m-commerce), mobile inventory etc.

A transaction is a set of operations that are performed completely or none of them, which is visible to other operations. This all-or-nothing feature is known as atomicity property where the commit protocols need to ensure atomicity and thus comprises a major issue in the execution of transactions. A distributed transaction forms a logical unit of job distributed over various mobile and fixed devices, such as money transfer from one bank account to another. Obviously, the transactions required for many mobile applications needs to maintain data consistency distributed over various sites. Transactions may also involve multiple mobile devices, besides fixed ones, as full participants such as in mobile commerce and mobile inventory.

Mobile environments are characterized by constraints such as, low power battery, disconnection, lower bandwidth, intermittent connection, low processing capacity of mobile devices. There are existing commit protocols that are designed for fixed networks, such as the traditional two-phase commit (2PC) protocol [7], unsuitable for mobile environments. Therefore, a number of commit protocols, such as UCM [1], TCOT [4] and M-2PC [4] have been developed to address these constraints. Unfortunately, existing approaches either only consider mobile hosts as initiators and not as full participants [3], or work only under strong assumptions, such as the uniformity of database systems [2] or the simultaneous connectivity of all mobile participants at the initiation of the transaction [4]. In addition, the existing protocols typically consider only a small subset of failures in the mobile environment. These weaknesses limit the applicability of existing approaches and cause new challenges for the design of efficient and reliable single phase commit protocols for diverged mobile environments. This paper proposes one phase, reliable, efficient and non blocking atomic transaction commit protocol called "Single Phase Reliable Timeout Based Commit protocol (SPRTBCP)".

The paper is organized as follows. In Section 2, we present the detailed architecture of proposed SPRTBC Protocol, including the notations and sequence of steps used in the protocol. Section 3 illustrates the transaction execution and recovery algorithm. Section 4 describes the performance evaluation and comparison carried out with the related protocols. Section 5 concludes the paper and outlines our future work.

II SINGLE PHASE RELIABLE TIMEOUT BASED COMMIT PROTOCOL (SPRTBCP)

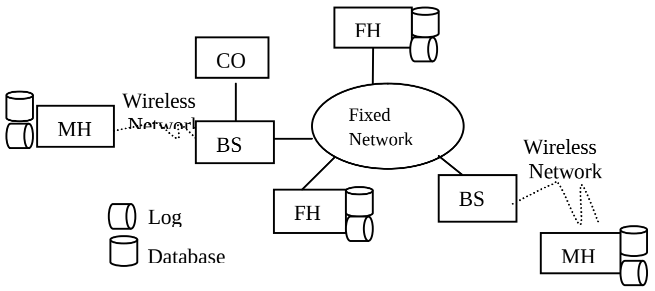
The Single phase reliable timeout based commit protocol is a non-blocking protocol and it preserves site autonomy. During normal execution the transaction is committed using single phase (eliminates voting phase of 2PC). The protocol employs timeouts to eliminate the blocking. The protocol assumes the participating data bases are ruled by a precise concurrency control mechanism and recovery algorithm. Non-blocking is achieved by using a broadcast primitive to deliver the decision messages. The protocol preserves site autonomy by enabling the participants to recover from failures independently. Fig. 1 illustrates the typical architecture of Single Phase Reliable Timeout Based Commit Protocol (SPRTBCP).

A. Protocol description

The basic idea of SPRTBCP is to eliminate the voting phase of the 2PC by introducing the properties of the local databases. In this context, a transaction is initiated by the TM-MH (Transaction Manager at MH) and this transaction is assured to be committed in a failure free environment by distributing the fragments at various participants (Part-FH's & MH's). When the acknowledgments for all fragments of a transaction T_i are received by the TM-MH, it means that the transaction fragments i.e. $e_{i0}, e_{i1}, e_{i2,\ldots,} e_{in}$ have been successfully executed till completion, TM-MH submits its positive commit message to the CO which can directly ask each participant host

DOI: 10.5176_2010-2283_1.4.101

accessed by the transaction T_i to commit, with no synchronization between the sites.



CO: Coordinator, BS: Base Station, FH: Fixed Host, MH: Mobile Host

Figure 1. Typical Architecture of Single Phase Reliable Timeout Based Commit Protocol

If a transaction fragment, say e_{ik} is aborted by participant$_k$ during its execution for any problem, the CO simply asks each accessed participant to abort that transaction. Assume that Participant$_k$ crashes during the one-phase commit of transaction T_i during which T_i may have been committed at other hosts. To guarantee T_i atomicity, the effects of the transaction branch e_{ik} have to be forward recovered in Participant$_k$.

The participants executing their respective fragments launch P*ack* and also update their local logs that contain physical redo log records generated during the execution of this operation along with the respective *log sequence number* (LSN). The CO registers the commit decision in its own log. Once Participant$_k$ recovers from its crash, it redoes set of operations using local log records with highest LSN and reinstalls them in the database.

To enforce transaction atomicity with site autonomy, SPRTBCP utilizes logging schemes introduced in their respective participants' database systems. On each participant site, local logs keep up each operation sent to it before its execution. During the *decision* phase, when a participant receives the *commit* decision, it updates the local database. If the local database crashes before completing the commit, it will abort the transaction. After the database recovery, the Participant re-executes all operations found in its log and belonging to the globally committed transaction. This approach guarantees global atomicity while preserving site autonomy**.** To achieve high performance and throughput, transactions are to be interleaved and executed concurrently. We assume that the concurrent executions of transactions are coordinated such that there is no interference among them.

In order to recover from failures, SPRTBCP maintains logs locally with each of the participants. Indeed, maintaining the logs locally, the CO must guaranty that the decision must be force written in stable storage before broadcasting its decision. In case of a participant crash during the one-phase commit, the failed transaction branches will be re-executed due to the operations registered in their respective redo logs. Following are the notations used in our algorithm.

1. *T_i : Transaction to be initiated by the TM-MH that includes $e_{i0},e_{i1},e_{i2},e_{i3},\ldots,e_{in}$ set of fragments*
2. *exec(e_{i0}), exec(e_{i1}),………, exec(e_{in}) : fragment executions on their respective hosts*
3. *Pack (e_{ik}): Acknowledgement for successful execution of fragment e_{ik} at participant's host k*
4. *Nack(e_{ik}): Acknowledgement for unsuccessful execution of fragment e_{ik} at participant's host k*
5. *Commit$_i$: Decision for commit of transaction T_i sent from the TM-MH to the CO and broadcast it by the CO to all the participant hosts*
6. *Abort$_i$: Decision for an abort transaction T_i sent from TM-MH and broadcast it by the CO to all the participant hosts*
7. *Cack$_i$: Acknowledgment for execution with commit of T_i at participants' hosts*
8. *Aack$_i$: Acknowledgment for execution with abort of T_i at participants' hosts*
9. *MH$_1$, MH$_2$,….,MH$_n$: Participating Mobile hosts at wireless network; Part-FH$_1$, Part-FH$_2$,….., Part-FH$_n$: Participant Fixed Hosts at wired network.*
10. *E_{tk} : CPU time required to execute k^{th} fragment at Participant$_k$*
11. *E_t: Maximum CPU time required to execute among all the fragments*
12. *m: Message to send and receive from their respective hosts.*
13. *send(m): Primitive to send message m to all the participant hosts*
14. *receive(m): primitive to receive a given message m from all the participant hosts*
15. *$T_i \rightarrow T_j$ represents any dependency between T_i and T_j.*

Fig. 2 gives the sequence of executions carried out during the transaction processing and shows the series of operations scenario introduced by the SPARTBC protocol. At Step 1, the transaction manager (TM-MH) initiates and fragments transaction T_i into set of sub-transactions (e_{i0}, e_{i1}….e_{in}) and distributes these among various participants including MH.

The participants upon receipt of their respective fragments, before they start the execution, all force write into their individual log area. They compute and send the time stamp required for executing their respective fragments ($E_{t0,\ldots\ldots}$, E_{tn}) to the TM-MH. Once the execution of their respective fragment is completed successfully, the *Pack* message is sent to TM-MH otherwise *Nack* is sent.

The Transaction manager at MH, upon receipt of E_t's from all the participants, it calculates maximum time Et= Max ($E_{t0,,,,,,,,}E_{tn}$) and waits up to the maximum time (E_t) for acknowledgement. If the transaction manager does not receive the acknowledgement from any one of the participant before time expiry, then it decides to abort the transaction and sends the same message to the CO.

If the TM-MH receives *Pack* from all the participants before time expiry, this means that all the fragments of T_i have been successfully executed. Thus TM-MH decides to commit and force writes this commit message into its local log and also issues a commit request to the CO.

Step 2 represents the message (commit/abort) received from the TM-MH is registered in CO's log. Note that this registration is done by a force writes and are buffered in local log (main memory) area and do not generate blocking I/O. If the TM-MH issues a commit request to the CO, it can thus take the commit decision. First, the CO Non-force writes its commit decision on stable storage in the same step, and CO asks each participant to commit T_i by broadcasting the commit decision to all. When the participant hosts receives the commit decision by the CO, they update local databases for the execution. Each operation is acknowledged up to the MH.

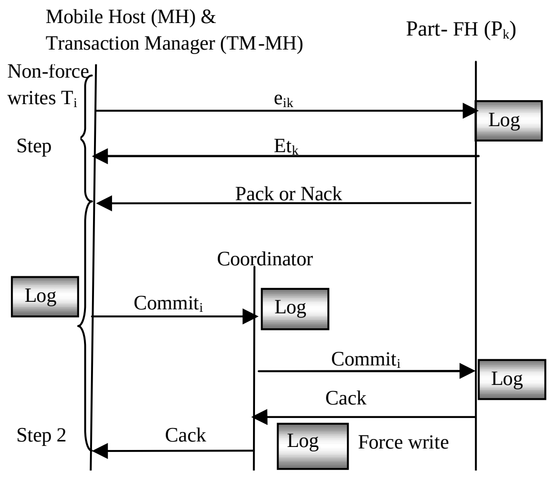


Figure 2. Single Phase Reliable Timeout Based Commit Protocol (SPRTBCP)

In order to be able to know whether or not a failed database has effectively committed a transaction and without violating its site autonomy, each participant maintains the local log containing the commit decision for T_i. This operation will be treated by the database as the other operations belonging to T_i, which is either all committed or all aborted atomically. Once the force written commit decision has been acknowledged by the participant$_k$ it asks the local database to commit transaction of e_{ik} fragment, as a local representative of T_i.

In a failure free environment, this commit always succeeds and is acknowledged up to the MH. For every transaction T_i, the TM-MH non-force writes each *Pack$_i$* in its log. When the acknowledgment is received from all the participants in the transaction, the TM-MH non-force writes an end$_i$ record. At this point, the TM-MH can forget transaction T_i. If transaction T_i is to be aborted, the TM-MH discards all T_i log records and broadcasts the abort message to the local databases, through its participants. SPRTBCP is a presumed abort protocol. Thus, abort messages are not acknowledged and the abort decision and transaction initiations are not recorded in the TM-MH log.

III. SINGLE PHASE RTBCP ALGORITHM

A. Transaction Execution Algorithm:

1) Transaction Manager at MH Algorithm

```
do forever {
  wait for (Initialization of transaction $T_i$ by the User)
  case($T_i$) of:
    beginTrans ($T_i(e_{i0},\ldots e_{in})$) at TM-MH // implements Step 1
    send $T_i$ ($e_{i0}, e_{i1}, \ldots e_{in}$) to MH and Part-FH's respectively ;
    wait for (receipt of $E_{t0},\ldots E_{tn}$ from their respective
    Participants)
    Compute Et = (MAX ($E_{t0}, E_{t1}, \ldots E_{tn}$)
    set time-out to $E_t$;
    wait for receipt of (Pack) from all or (Nack) from any
    participant or time-out ($E_t$)
    // any $T_i$'s operation after exec($e_{i0},\ldots, e_{in}$)
     if (Pack) from all then{
      force write $T_i$'s commit decision in local log; // implements
      Step 1 decides to Commit$_i$
     send Commit$_i$ to CO;}
    else if( (timeout) or receipt of an (Nack) from any of
     participant)
      { send Abort$_i$ to CO;} // implements Step 1}
   Wait for (receipt of ack for Decision message from CO)
   Case$_1$: Aack: call recovery procedure; // implements Step 2
   Case$_2$:Cack: Force write $T_i$ log records and $T_i$ commit
    decision in local log and update local DBMS; //
implements Step 2 }
```

2) MH Algorithm (Mobile host –Initiation of T_i)

```
do forever {
 wait for (receipt of message m for fragment $e_{i0}$ from TM-MH)
case (m) of
{ compute $E_{t0}$ and send $E_{t0}$ to TM-MH
set time-out to $E_{t0}$;
$e_{i0}$ is sent to local log;
exec($e_{i0}$) : execution of ($e_{i0}$) at MH;
if (MH decides to (Pack)) then
{ force write $e_{i0}$ log records and Pack decision in local log;
send Pack to TM-MH;
wait for(the reciept of a decision by CO)
goto 111}
else if (MH decides to (Nack) or (timeouts)
{send Nack to TM-MH;
 send Nack decision to local log and forget $e_{i0}$ ;}
111: // for any $T_i$'s operation ,
Case$_1$: If (decision message from CO is (Abort$_i$))
{ if ((Pack) sent by the MH ){ call recovery algorithm }
// implements Step 2
send Aack to CO and forget $T_i$. }
Case$_2$ : if ( decision message from CO is (Commit$_i$))
{ force write $e_{i0}$ log records and $T_i$ commit decision in local
log and update local DBMS;
send Cack to CO; // implements Step 2 }
```

3) Coordinator Algorithm

```
do forever {
 wait for (receipt of a decision for $T_i$ (Abort$_i$ or Commit$_i$) from
the TM-MH)
  Case$_1$ : {Abort$_i$: broadcast Abort$_i$ to all the participants ; //
implements Step 2}
  Case$_2$: {Commit$_i$: Force write $T_i$'s commit decision in local
log; broadcast commit$_i$ to all ; // implements Step 2}
wait for (receipt of an (Aack or Cack) from all)
 Case$_1$ : {Aack$_i$: send(Aack$_i$) to TM-MH; // implements Step 2}
 Case$_2$ : {Cack$_i$: send(Cack$_i$) to TM-MH ; // implements
Step2}}
```

4) Participant$_k$ algorithm

```
do forever {
 wait for (receipt of message m for fragment $e_{ik}$ from TM-MH)
case (m) of:
{compute $E_{tk}$: send it to TM-MH
set time-out to $E_{tk}$;
```

```
send ($e_{ik}$) to local log ; // implements Step 1
exec($e_{ik}$); execution of ($e_{ik}$) at their participant$_k$;
// implements Step 1
if (participant$_k$ decides to (Pack)) then
{ force write $e_{ik}$ log records and Pack decision in local log;
send Pack to TM-MH;
wait for(the reciept of a decision by CO)
goto 111}
else if ((timeouts) or (Nack) from any of the participant$_k$)
{send Nack;
send Nack decision to local log and forget $e_{ik}$ ;}
111 : // for any $T_i$'s operation
Case$_1$ :If (decision message from CO is (Abort$_i$))
{ if (Pack sent by the Participant$_k$) {call recovery algorithm;
 // implements Step 2
send Aack to CO and forget $T_i$;}
Case$_2$ :If( decision message from CO is(Commit$_i$))
{force write $e_{ik}$ log records and $T_i$ commit decision in local log
and update local DBMS;
send Cack to CO; // implements Step 2}
```

Figure 3. Algorithm for Single phase Reliable Timeout Based Commit (SPRTBC) Protocol

The Fig. 3 shows an algorithm for implementing the SPRTBC protocol. The Participants directly receives the decision of the TM-MH for a transaction Ti (Commit$_i$ /Abort$_i$) through CO i.e. on receipt of decision from TM-MH the CO informs the participants of a transaction of its decision through a simple broadcast primitive presented in Fig. 3(e). Hence, a participant hosts updates their local database and conforms to the decision using *Cack*/*Aack* primitives. It also contains recovery algorithm (Fig. 4), which is used to recover the database into consistent state in case of failures.

```
Pack($e_{ik}$)
{ upon receipt of any fragment ( $e_{ik}$) at Participant$_k$
{ If (successful execution of ( $e_{ik}$)) then
{send Positive acknowledgement for $e_{ik}$ to TM-MH }}}
```

Figure 3 (a) Positive acknowledgement primitive

```
Nack($e_{ik}$)
{ upon receipt of any fragment ( $e_{ik}$) at Participant$_k$
{If (unsuccessful execution of ($e_{ik}$) or (timeout) or
(disconnected)) then
{send Negative acknowledgement for $e_{ik}$ to TM-MH}}}
```

Figure 3(b) Negative Acknowledgement primitive

Fig. 3(a) shows *Pack (Positive acknowledgement) primitive* is an acknowledgement for successful execution of fragment e_{ik} at Participant$_k$. Every participant host sends the positive acknowledgement to designate the fragment e_{ik} is executed successfully. This guarantees that all participant hosts will eventually deliver *Pack to TM-MH.*

Fig. 3(b) shows *Nack (Negative Acknowledgement) primitive* is an acknowledgement for unsuccessful execution of fragment e_{ik} at Participant$_k$. If any participant host failed to complete the transaction execution then it sends negate acknowledgement message to indicate the fragment e_{ik} is not executed successfully. This guarantees that all correct participant hosts will finally deliver N*ack to TM-MH*.

```
Cack($e_{ik}$)
{      upon (receipt of Commit$_i$ from CO) at Participant$_k$:
        If (Force write for $e_{ik}$ is successful)
       {send an acknowledgement for commit: Cack ($e_{ik}$) to
CO}}
```

Figure 3(c) Commit acknowledgement primitive

```
Aack($e_{ik}$)
{ upon (receipt of Abort$_i$ from CO) at Participant$_k$
   { If (Force write of $e_{ik}$ is unsuccessful) then
     { Send an acknowledgement for abort: Aack($e_{ik}$) to CO}}}
```

Figure 3(d) Abort acknowledgement primitive

Fig. 3(c) shows Commit acknowledgement primitive is an acknowledgment by the Participant$_k$ on the receipt of Commit$_i$ from the CO. Fig. 3(d) shows Abort acknowledgement primitive is an acknowledgment by the Participant$_k$ on the receipt of Abort$_i$ from CO.

```
Broadcast Primitive (broadcast a message to all the
participants)
Broadcast (m) to all
 {// the CO executes:
   send(m) to all participants }
```

Figure 3(e) A Broadcast primitive

Fig. 3(e) shows the broadcast primitive, that sends the messages to all the participants.

B. Recovery algorithm

1) Participant's Algorithm – in case if participant crashes

```
Participant algorithm
Case 1: If (Participant$_k$ crashes before sending (Pack($e_{ik}$)) to
TM-MH {
 if (Participant$_k$ recovered before (time expires) AND force
written Pack($e_{ik}$) into its local log)
{Send Pack($e_{ik}$)}
else if (Participant$_k$ not recovered before (time expires) OR
force written Nack($e_{ik}$) into its local log)
{backward recover the transactions that reached their commit
state before the crash
Send Nack($e_{ik}$)}}

Case 2: If (Participant$_k$ crashes after sending (Pack($e_{ik}$)) to
TM-MH and before getting Commit$_i$ from CO)
{ Once participant$_k$ recovered
  { Wait for CO's Decision
    While waiting for CO's decision
    { If (timeouts) {resend Pack ($e_{ik}$) to TM-MH}
       else { (upon receipt of Commit$_i$ from CO before time
       expires : send Cack to CO }}}}
     else If (Participant$_k$ crashes after sending (Nack($e_{ik}$)) to
            TM-MH and before getting Abort$_i$ from CO)
{ forget the $e_{ik}$}
```

```
Case 3: If( Participant_k crashes before sending Cack(e_ik)) to
CO{ If commit_i ∈ local log { send Cack(e_ik)}}
Else If (Participant_k crashes before sending (Aack(e_ik)) to CO
If {Abort_i ∈ local log {send Aack(e_ik)
 Forget T_i}}

Case 4: If (Participant_k crashes after sending Cack(e_ik) or
Aack(e_ik)) to CO {Forget the transaction of e_ik}
```

Coordinator's algorithm

```
For CASE 2:
 for each transaction T_i in which Participant_k participates
{ if CO receives duplicated Pack(e_ik)
{ if ((Commit_i) ∉ coordinator's log), then
{ send (Abort_i) to the requested participants and forget T_i}
else  if( (Commit_i) ∈ coordinator's log) then
{ send (Commit_i) to the requested participants }}}

2) Coordinator's algorithm (in case if coordinator crashes)
CASE 1: If CO crashes (after receiving (Commit_i / Abort_i) from
TM-MH or before broadcasting (Commit_i / Abort_i) to all
For each transaction T_i
 { if ((Commit_i) ∈ coordinator log) then
{Decision =Commit_i} Broadcast Commit_i to all}
else if((Abort_i) ∈ coordinator log) then
{Decision =Abort_i} Broadcast Abort_i to all}}

CASE 2: If CO crashes (after broadcasting (Commit_i / Abort_i)
to all) { Wait for acknowledgement from all
While waiting
{if ((timeouts) and (Commit_i) ∈ coordinator log) then
{Broadcast Commit_i to all}}
Expect for new transaction}
```

Figure 4. Recovery Algorithm for Single phase RTBCP Protocol

IV. PERFORMANCE EVALUATION

In this section, the performance of SPRTBCP is compared with 2PC, UCM, TCOT respectively. 2PC is the most well-known *blocking* commit protocol, while UCM is one phase commit protocol that has been proposed for light weight processing, and TCOT is timeout based non-blocking commit protocol hence we compare SPRTBCP with the above protocols.

Let *n* denote the number of participants in the transaction. The other parameters we have considered to analyze the performance are the time delay, number of force writes and message complexity that needs to broadcast the decision message.

In SPRTBCP, the CO broadcasts the decision using the broadcast primitive presented in Fig. 3(e), where the message complexity is (n+1) and the time delay is ∂+e (e is no of timeout extensions). In 2PC, we need to add the complexity of the vote request and vote collection message rounds.

The results illustrated in Table 1 gives comparison of SPRTBCP with 2PC, UCM, TCOT in terms of time delay, number of force writes and message complexity. The SPRTBCP is a presumed abort protocol, thus the MH uses the broadcast primitive to broadcast an *Abort* decision when the MH does not receive the acknowledgement of transaction execution before time expires, hence we can prove that it is non blocking protocol.

Table I Performance Metrics

Commit Protocol	No. of phases	Message Complexity	Latency	No. of force writes
2PC	2	4n	3∂	2n+1
UCM	1	2n	∂	n+1
TCOT	1	(2n-1) +e	∂ +e	n+1
MRTBCP	1	(2n-1) + e	∂ + e	n+1
SPRTBCP	1	(2n-1) + e	∂ +e	n+1

A. Simulation Model

We present the simulation model and results of experiments to evaluate the performance of the SPRTBCP protocol. With the simulation model, the transactions are generated and fragmented by the Transaction Manager (TM-MH) at MH, at each simulation run; at least 5 transactions are generated according to the parameters of the transaction model to simulate the features of mobile computing such as frequent disconnection and long-lived transaction operations. They are submitted to the execution model for further processing. Deadline of each transaction is also performed by the TM-MH. The transactions generated at TM-MH are fragmented into set of sub transactions and part of each transaction is processed by Mobile Execution Unit (MEU) available at MH. The remaining fragments are sent to the Fixed Execution Unit (FEU) at fixed host. The FEU executes and sends resultant message to the TM-MH. Based on the results from each participant the TM-MH decides to commit / abort the whole transaction. Hence the decision is forwarded to the CO. The CO broadcast the same decision message to all the Participants and waits for their acknowledgements. MS is responsible for the transmission of messages between hosts. Table 2 summarizes our simulation parameters that are used to state the system resources and other overheads.

B. Results and Discussions

For the evaluation of the SPRTBCP, the main performance metric considered is Average Commit Time; that gives the number of committed transactions per second and the other metric is effect on throughput with increase in number of transactions and with an extended time requests. Fig. 5 shows the effect of Average Commit Time on TCOT, MRTBCP and SPRTBCP. It can be observed that, as the number of transactions increases the commit time taken is also increased. The commit time is the time taken to complete the execution of a transaction per unit time.

Fig.6 illustrates the effect of E_t change on Throughput. The throughput is defined as the number of successfully committed Mobile Transactions per unit time (in msec). With each handoff, fixed amount of delay was added to the execution time as E_t of a fragment. E_t =10 means 10% of workload ask for the extension of E_t.

For some fragments the extension were denied and some did not ask for any extension [8]. We can observe that the effect of E_t is not that severe on the throughput. This could be further improved by reducing the amount of extension requests by carefully evaluating the initial value of E_t for each fragment

TABLE II SIMULATION PARAMETERS

Parameter		Value
FHosts	Fixed hosts	03
Mhosts	Mobile host	02
DBSize	No. of tuples in database	30 Tuples
Read/Write Ratio	No. of Read & Write operations	70:30
TrgnTm	Time in seconds to generate the transactions.	0 sec
DissconProb	Probability of getting disconnected MH	0.001 msec
Failurprob	Probability of having weak wireless link	0.001 msec
MCPUTime	Avg CPU time to process a Fragment at MH	50 msec
FCPUTime	Avg CPU time to process a fragment at FH	50 msec
MsgCPUTime	Avg CPU time to process a Message at FH	02 msec
NumfragMT	Number of fragments / transactions	05
TransdelayFH	Transmission Delay over Wired network	10 msec
TransdelayMH	Transmission Delay over Wired network	5 msec

Fig.7 illustrates the message complexity, the message complexity is defined as the number of messages sent and received in average by each mobile host during the execution of the Mobile Transaction. the results shows that the number of message rounds with 2PC, TCOT, MRTBCP & SPTBCP is almost same, they use one phase to commit the transactions, but with TCOT it may need to have more number of messages and due to Two Phase Commit operation of 2PC it needs to have huge round of messages.

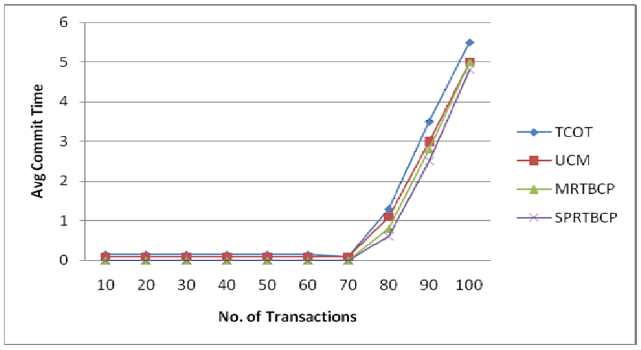


Figure5. Average Commit Time V/S No. of transactions

V. CONCLUSION

Non-blocking ACPs normally achieves a higher latency. Mobility and portability pose new challenges to the management of mobile database and distributed computing. Existing ACPs need to be upgraded to adapt them to the new environment. This paper proposes an atomic commitment protocol, called as SPRTBCP offers the non-blocking property with one phase commit operation having lesser message complexity and also preserves site autonomy making it compatible with inherent databases. Our protocol is aimed at handling new challenges including site failures and message loss blocking-free manner. Compared to MRTBCP, it is single phase commit protocol without the log agent, due to which the it reduces average commit time and message complexity leading to an overall cost reduction of wireless communication. Hence with the use of SPRTBCP it can be proved that, we can achieve better performance and reliable execution of transactions in case of disconnections, failures and also during mobility as compared to the existing ACPs like TCOT, UCM, M-2PC etc.

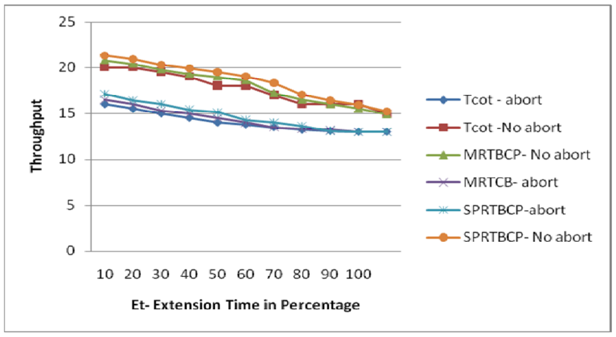


Figure 6. Effect of E_t on throughput

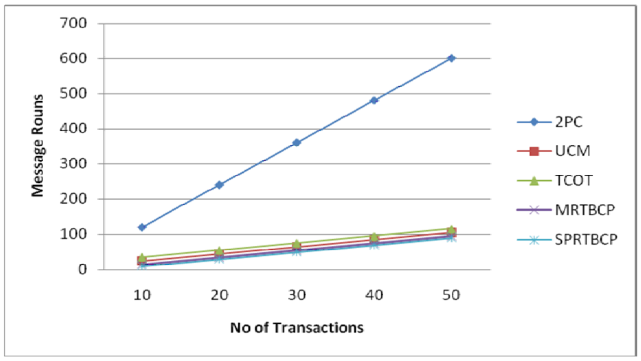


Figure7. Message Complexity

REFERENCES

[1] C. Bobineau, P. Pucheral, and M. Abdallah. "A Unilateral Commit Protocol for Mobile and Disconnected Computing", In *PDCS*, USA, 2000.

[2] B. Bose, S. Sane, "Distributed Timeout Based Transaction Commit Protocol for Mobile Database Systems," in Proceedings of International Conference and Workshop on Emerging trends in Technology (ICWET -2010),2010, pp. 518-523.

[3] Christophe Bobineau, Cyril Labb'e, Claudia Roncancio, Patricia Serrano - Alvarado, "Comparing Transaction Commit Protocols for Mobile Environments," in Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04) 1529-4188/04 IEEE.

[4] V. Kumar, N. Prabhu, M. H. Dunham, and A. Y. Seydim, "TCOT – A Timeout-Based Mobile Transaction Commitment Protocol," *IEEE Transactions on Computers*, 51(10), 2002.

[5] P. Serrano, C. Roncancico, M. Adiba, "A Survey of Mobile Transactions," DAPD Jnl., 16(2), 2004.

[6] M. M. Goreyand , R. K. Ghosh, "The Recovery of Mobile Transactions," in Proceedings of the 11 th International Workshop on Database and Expert Systems Applications (DEXA'00) 2000 pp. 23, IEEE.

[7] N. Nouali, A. Doucet, and H. Drias. A two -phase commit protocol for mobile wireless environment. In H. E.Williams and G. Dobbie, editors, *Sixteenth Australasian Database Conference (ADC2005)*, volume 39 of *CRPIT*, pages 135–144, Newcastle, Australia, 2005. ACS.

[8] B. Harsoor, S.Ramachandram, "Reliable Timeout Based Commit Protocol," Proceedings of 2nd International Workshop on Trust Management in P2P Systems (IWTMP2PS-2010) CNSA-2010, Springer Verlag 2010, pp. 417-423.

[9]B. Harsoor,S. Ramachandram, " Modified Reliable Timeout Based Commit protocol " proceedings of Eighth International conference on wireless and optical communications networks (WOCN -2011) IEEE

Authors

Mrs. Bharati Harsoor, received her bachelor's degree in CSE (1995), Masters in Computer Science (2001). She is a Research Scholar at Osmania University, Hyderabad. She is presently working as Asst Professor, Department of Information Science and Engineering, Gulbarga, Karnataka State, India and published m any papers in various national and international conferences and journals. Her areas of interest are Mobile Computing, Databases, and Software Engineering. She is member of Institution of Engineers and Telecommunication Engineering (IETE)

Dr. S. Ramachandram (1959) received his bachelor's degree in Electronics and Communication (1983), Masters in Computer Science (1985) and a Ph.D. in Computer Science (2005). He is presently working as a Professor and Head, Department of Computer Science, University College of Engineering, Osmania University, Hyderabad, India. His research areas include Mobile Computing, Grid Computing, Server Virtualization and Software Engineering. He has authored several books on Software Engineering, handled several national & international projects and published several research papers at international and national level. He also held several positions in the university as a Chairman Board of Studies, Nodal officer for World Bank Projects and chair of Tutorials Committee. He is a member of Institute of Electrical and Electronic Engineers (IEEE), Computer Society of India (CSI) and Institute of Electronics and Telecommunication Engineers (IETE).

,