A Primary Shift Protocol for Improving Availability in Replication Systems

Almetwally M. Mostafa
CCIS, King Saud University
Riyadh, KSA and
Faculty of Engineering
Alazhar University, Cairo, Egypt

Ahmed E. Youssef
CCIS, King Saud University
Riyadh, KSA and
Faculty of Engineering
Helwan University, Cairo, Egypt

ABSTRACT
Primary Backup Replication (PBR) is the most common technique to achieve availability in distributed systems. However, primary failure remains a crucial problem that threatens availability. When the primary fails, backup nodes in the system have to elect a new primary node in order to maintain adequate system's operation. During election, the system suffers from transaction loss, communication overhead due to messages exchange necessary to preserve data consistency, and a notable delay caused by the execution of Leader Election Algorithms (LEA). Primary failures can be unpredictable (i.e., unplanned), such as primary node crashes and network outages, or predictable (i.e., planned), such as primary's scheduled shutdown to perform routine maintenance or software upgrade. Traditionally, PBR employ LEA to recover from both unplanned and planned outages. In this paper, we propose a novel protocol, called Primary Shift Replication (PSR), to avoid election during planned outages. PSR shifts the primary role from the current primary to another scheduled node (without election) when a planned outage is about to occur. Number of messages and communication time required to shift the primary node to another node is much less than number of messages and time required to perform leader election; therefore, PSR improves system's availability. Moreover, PSR guarantees no transactions loss during the shift mode, hence, it preserves data consistency.

General Terms
Distributed Systems, Algorithms.

Keywords
Primary Backup Replication, Leader Election, Transaction Store, Replication Systems.

1. INTRODUCTION
Replication is the heart of today's emerging information technologies such as cloud computing, mobile computing, and pervasive computing. Primary Backup Replication (PBR) approach is the most common practical technique used for several distributed applications such as distributed databases, distributed white board, collaborative applications, and distributed agenda that require a high degree of consistency and availability of shared data objects. PBR approach requires one node (i.e., leader or primary) to act as an organizer for other replicas in the distributed system [1, 2, 12, 16, 17, 18]. The primary maintains object store consistency by controlling access to the shared objects and executes the transactions that clients submit at different replicas. In Chain Replication (CR) [16], the replicas are organized in a chain with a head (node with the maximum ID) and a tail (node with the minimum ID), the head node plays the role of the primary in case of update transactions while the tail node responds to query requests, this alleviates the load on the primary node.

The availability of replication services depends heavily on the availability of the primary node or the leader. If the leader fails for any reason, another node should be elected as a leader to maintain adequate system's performance. Leader Election Algorithms (LEA) [3-6,13-15] are employed to elect a new leader to substitute the failed leader. The new elected leader must acquire information on the shared objects such as the lock table at the primary and the most recent objects' states. However, acquiring such information is a challenging task since the previous leader crashed without delivering its own lock table and the last updated objects' states to the new elected leader. The time taken by LEA to elect a new leader is an idle time during which there are no active transactions. In addition, the number of messages that nodes exchange to preserve object store consistency during election linearly increases with the number of nodes and objects in the system which causes a considerable communication delay. Obviously, these factors negatively affect the availability of the replication system. Moreover, the transactions that are issued just before primary failure may be lost (not committed nor aborted), since nodes are busy collecting information on objects that have been locked by the failed primary. This may render shared objects inconsistent.

In distributed systems, leader failure is not only limited to unplanned (unpredictable) crash cases such as power off and network outages. There are several planned primary downtime cases for scheduled backup, software upgrade, or routine maintenance [9]. For example, in [19] Amazon Web Service (AWS) reported the following primary outage: "*At 12:47 AM PDT on April 21st, a network change was performed as part of our normal AWS scaling activities in a single Availability Zone in the US East Region. The configuration change was to upgrade the capacity of the primary network. During the change, one of the standard steps is to shift traffic off of one of the redundant routers in the primary EBS network to allow the upgrade to happen. The traffic shift was executed incorrectly and rather than routing the traffic to the other router on the primary network, the traffic was routed onto the lower capacity redundant EBS network".* AWS used a sophisticated network solution (i.e., shifting traffic) to replace the primary network by a secondary one. However, due to incorrect traffic shift a large number of nodes lost connection to their replicas leaving many nodes stuck in a loop. We believe that the problem should have been addressed at the middleware level where replicas can cooperate to transfer the load from one region to another without relying on solutions

at network level.

Traditionally, PBR systems employ LEA to recover from both planned and unplanned outages. However, we believe that planned outages can be managed in a different way that avoids election drawbacks. In this paper, we propose a novel protocol, Primary Shift Replication (PSR), that avoids election in planned outages by shifting the primary role from the current primary to another scheduled node whenever a planned outage is about to occur. In PSR, the administrator feeds the system with a schedule for the planned outages where primary shift should take place. This gives administrators more flexibility to schedule backup, upgrade software, and perform routine maintenance for the primary node. The shift time is too short compared to the election time; therefore, system's availability is improved. Moreover, PSR guarantees no transactions loss during the shift mode, hence, it preserves consistency. If an unpredictable failure occurred during primary shift, election is employed to tolerate the failure. To the best of our knowledge, there is no work in literature that adopts primary role shift in replication systems.

In order to test the feasibility of our approach, we developed a prototype for the replication system and conducted a set of experiments to show the merits of our approach. The results are very promising in terms of avoiding election drawbacks, improving availability, and preserving consitency. The rest of this paper is organized as follows: in section 2 we review the PBR approach, in section 3, we describe our proposed primary shift protocol, in section 4, we present experimental results, and in section 5, we give our conclusions and future work.

2. PRIMARY BACKUP REPLICATION

A replication system is composed of a set of fully connected replicas (nodes) with one replica assigned exclusively as a primary. Each replica serves a set of clients and each client can connect to only one replica at a time. In PBR approach, the primary replica accepts and executes client requests for the entire data objects and propagates the results to other replicas in the system. The enhanced and light-weight PBR approach [7,8] allows all replicas to share extra workload by executing client's requests. However, the primary continues to play a significant role by holding the permission for accessing the shared replicated objects. The primary exclusively holds a lock table to coordinate among replicas for the purpose of concurrency control.

In a replicated object store, the object is any piece of data being shared such as counters, rectangles, files, DB records, etc [11]. Each replica maintains the following:

- ***membership:*** a set of references to the connected replica, S= {s_1, s_2,…, s_n}
- ***local_store:*** a set of references to the replicated objects, O = {o_1,o_2, …,o_m}
- ***primary:*** a reference to the current primary replica, i.e., *primary*=self or s_i
- ***transaction*:** $T_{tid}(O_{tid},M_{tid})$ is a procedure that uses a set of methods M_{tid}={m_1,…,m_k}, to modify the states of a subset of object O_{tid}={o_1,..,o_z}, where tid refers to transaction ID. A transaction state can be active (under processing), commit (finish processing), or abort (cancel processing).
- ***lock_table*:** is a three-field table at the primary node containing *object_reference* (o_i) which refers to the object requested to be locked, *replica_reference* (s_i) which refers to the replica requested to lock the object, and *tid* of the active transaction.
- ***primary time stamp***: initially $T_{ptstamp}$ =1

In a replication system, each replica is a state machine whose transitions are triggered by the reception of an event. As described in [10], events may carry information such as a data message or a group membership in one or more attributes. An event is denoted by <EventType | att1, att2, . . >. We consider two types of events:

- ***Request:*** refers to a service requested by a client from a replica such as *add_object* and *submit_transaction* or by a replica from another replica such as *lock_object.*
- ***Notification:*** refers to a response to a request from replica to replica such as *ok_lock/ko_lock* or from replica to client such as and *trans_abort* and *trans_commit*.

The behavior of a replication system is controlled by a protocol which includes a set of rules; each rule is triggered by an event. These rules are described in the following subsections.

2.1 Create New Object

Upon receiving *add_object* request from a client, the underlying replica creates a new local object in its local store and triggers other replicas to create copies of this object in their local stores. All copies (i.e., object replicas) are considered as one logical object in the system and is identified by a unique ID. Rules 1a and 1b shown below describe the object creation process. The set of actions taken during this process are the following: 1) Client, C_i, trigger a replica to create a new object, 2) The triggered replica adds new *object_reference* (o_i) to its local store (O), 3) The replica notifies its clients with object reference, 4) The replica triggers all other replicas by *add_object* request, 5) Actions 2 and 3 are repeated at each other replica.

Rule 1a- object creation from client to replica

Upon event < $add\_object()|class, o_i \mid C_i$ > do
  o_i = *new class*
  $O = O \cup \{o_i\}$
  client ***Notify*** <new, o_i>
  $\forall s_i \in S$
    Notify < $add\_object()|class, o_i \mid self$ >

Rule 1b: object creation from replica to a replica

Upon event < $add\_object()|class, o_i \mid S_i$ > do
  o_i = *new class*
  $O = O \cup \{o_i\}$
  client ***Notify*** <new, o_i>

2.2 Submit Transaction

When a client submits a transaction to its connected replica, the replica takes the following actions: 1) Client, C_i, triggers the replica to submit a transaction, 2) The replica creates a

unique transaction ID (tid) and add the transaction to the *active_list*, 3) The replica identifies *object_references* (O_{tid}) to be locked, 4) The replica sends *lock_objects* request to the primary. These actions are listed in rule 2.

Rule 2: transaction request from client to replica

Upon event $< trans() \mid proc(), O, M, C_i >$ do
$create\ \ new\ tid$
$active\_list = active\_list \cup tid\ \ \#proc()$
$primary\ \boldsymbol{Trigger} < lock() \mid O, S_i, T_{tid} >$

2.3 Lock Request

When the primary node receives the *lock_object* request from the replica, it takes the following actions: 1) The primary receives *lock_object* request from a replica, 2) The primary checks to see if any object in O_{tid} is already locked, 3) The primary responds by *ko_lock* if an object is already locked, 4) The primary responds with *ok_lock* and updates the *lock_table* with *object_reference* (o_i), *replica_reference* (s_i) and *tid*. if all objects are free. These actions are described in rule 3.

Rule 3: lock request at primary

Upon event $< lock() \mid O, S_i, tid >$ do
$T_{ptstamp} = T_{ptstamp} + 1$
$flag = true$
$\forall\, o_i \in O\ \ if\ \sqsubseteq o_i \mid \epsilon\ lock\_table\ then\ flag = false$
$if\ (flage = false)$
$\quad S_i\ \boldsymbol{Notify} < ko\_lock() \mid tid, T_{ptstamp} >$
$else$
$\quad \forall o_i \in O\ \ lock\_table = lock\_table \cup \{< o_i, S_i, tid >\}$
$S_i\ \boldsymbol{Notify} < ok\_lock() \mid tid, T_{ptstamp} >$

2.4 Execute transaction rule

Upon receiving *ok_lock* from the primary, the underlying replica executes the following set of actions 1) Replica executes transaction procedure and updates the object in its local_store, 2) Replica triggers other replicas to update objects in their local_store, 3) Primary release locked objects, 4) Replica adds tid to *commit_list* and remove it from *active_list*, 5*)* Replica notifies client with *trans_commit* message. These are actions are shown in rule 4a.

On the other hand, upon receiving *ko_lock*, the underlying replica takes the following actions: 1) Replica adds tid to *abort_list* and removes it from *active_list*, 2) Replica notifies client with *trans_abort* message. These are actions are clarified in rule 4b.

When the primary receives *update_object* notification, it releases object locks from the *lock_table* as shown in rule 5.

Rule 4a: ok_lock event

Upon event $< ok\_lock() \mid , T_{ptstamp} >$ do
$commit\_list = commit\_list \cup tid$
$proc() = return\ P\ (\forall\ tid\ \#\ P \in activetans)$
$active\_list = activet\_list - tid\#P$
$commit\_list = commit\_list\ \cup tid\ \#\ T_{ptstamp}$
$execute\ Proc()$
$\forall\ s_i \in S$
$\boldsymbol{Trigger} < update() \mid proc(), self, tid, T_{ptstamp} >$
client ***Notify*** <tid, commit>

Rule 4b: ko_lock event

Upon event $< ko\_lock() \mid , T_{ptstamp} >$ do
$active_{list} = active_{list} - tid\#P$
$abort\_list = abort\_list \cup tid\#\ T_{ptstamp}$
$client\ \boldsymbol{Notify} < tid, abort >$

Rule 5: update event

Upon event $< update() \mid proc(), s_i, tid, T_{ptstamp} > do$
$if\ primary\ \forall o_i\ \epsilon O\ lock\_table = lock\_table - \{o_i, \_, \_\}$
$commit\_list = commit\_list\ \cup tid\ \#\ T_{ptstamp}$
$execute\ proc()$
$client\ \boldsymbol{Notify} < tid, commit >$

3. THE PRIMARY SHIFT PROTOCOL

We propose a novel approach that avoid election in planned outages by shifting the primary role from the current primary to another scheduled node whenever a planned outage is about to occur. At anytime, there could be a primary outage; these outages are defined by the administrator to support operations such as scheduled backup, software upgrade, or routine maintenance for the primary node. Table 1 shows some of these outages, outage #1, for example, implies that if server (s_1) was the primary at the time (12:30, 12/5/2012) then it should be replaced by server s_2, because s_1 is subjected to software upgrade. When the shift mode starts at t=T_s, the old primary, s_1, immediately forwards the *lock_table* to the new primary, s_2, and every other replica changes the primary identity from s_1 to s_2.

Table 1. a schedule of planned outages

outage #	outage time (T_s)	Current primary	New primary
1	12:30, 12/5/2012	s_1	s_2
2	5:15, 5/8/2012	s_2	s_3
3	7:45, 12/10/2012	s_3	s_4

As figure 1a illustrates, the bolded horizontal line indicates the current primary for the system. At the outage time, T_s, each node enters the shift mode, the transfer time of the *lock_table* is given by (T_l-T_s) where T_l denotes the time at which the *lock_ table* is completely received by the new primary. The the shift mode time is given by (T_n-T_s) where T_n denotes the time at which all replica return to the normal mode. The *lock_ table* transfer time (T_l-T_s) varies depending on the network speed between replicas and the data size in the *lock-table* itself. The time (T_n-T_l) is the time needed by the new primary to unlock the objects exist in the *lock-table* delivered by the old primary. This time depends on the arrival rate of update notification messages.

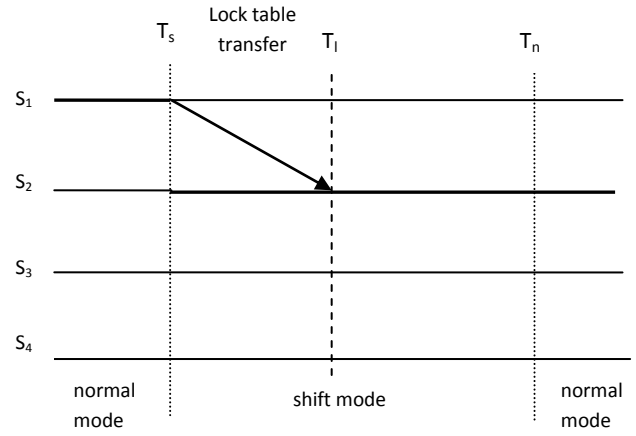

Fig. 1a: X-primary (s_1) sends local lock_table to new primary (s_2)

During the shift mode time ($T_s<t<T_n$), any new transaction issued by a client is aborted, however, there are some old transactions that are in progress, (i.e., transactions started before T_s and were being handled by any replica), these transactions continue in the shift mode and are completed by cooperation between the old and the new primaries without any transaction loss as we will explain in the following cases:

Case 1: When there are lock requests which were issued before entering the shift mode ($t<T_s$) from replicas, such as s_3 and s_4 in figure 1b, and arrived at the old primary, s_1, during the shift mode ($T_s<t<T_n$), these requests are forwarded from the old primary, s_1, to the new primary, s_2,. At s_2 there are two possible situations:

1) The lock request arrives at s_2 before transferring the *lock_table* ($t=T_1<T_l$), in this situation, s_2 stores this request in a queue until the *lock_table* is completely received.
2) The lock request arrives at s_2 after transferring the *lock_table* ($t= T_2 > T_l$), in this situation, s_2 handles this request according to the rules presented in the previous section (subsection 2.3, rule 3).

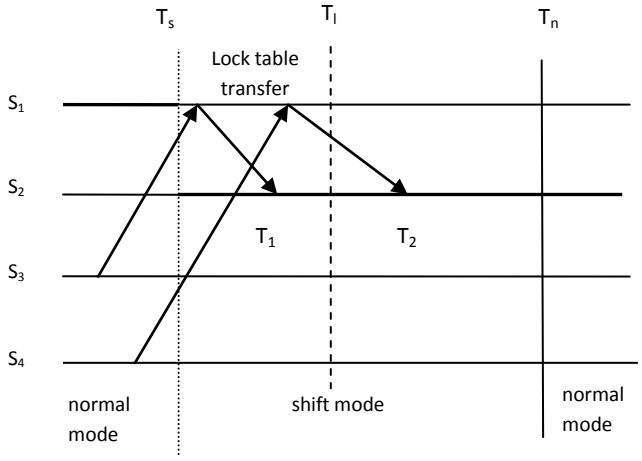

Fig. 1b: lock request issued before shift mode starts and arrived before/after lock_table transfer completed

Case 2: recall section 2, when an update notification arrives at a replica, it updates the underlying objects in its *local_store*. If the update notification arrived during the shift mode, there are two possible situations:

1) The update notification message arrived at the new primary, s_2, before finishing *lock_table* transfer ($t<T_l$) as illustrated in figure 1c. In this case, s_2 stores the update notification in a queue until the *lock_table* is completely transferred. When the *lock_table* is transferred, s_2 calls the update notifications in the queue, performs the necessary updates and releases the object locks.
2) The update notification message arrived at s_2 after transferring the *lock_table* ($t>=T_l$) as depicted in figure 1d. In this case, s_2 directly performs the update notification and releases the object locks.

The above cases are repeated until the new primary satisfied two conditions:

1) The *lock_table* is empty, which means that no locks are granted to any replica for any transaction.
2) All pending locks and updates requests in the *request_queue* are served and the queue is empty.

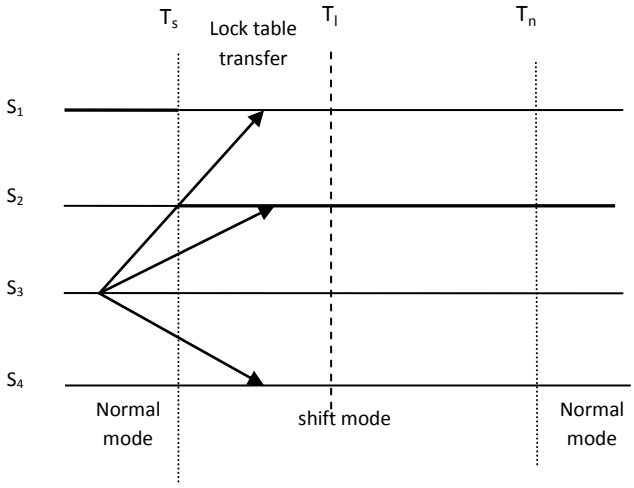

Fig. 1c: s_3 sends update notifications before shift mode starts and notifications arrived before the arrival of

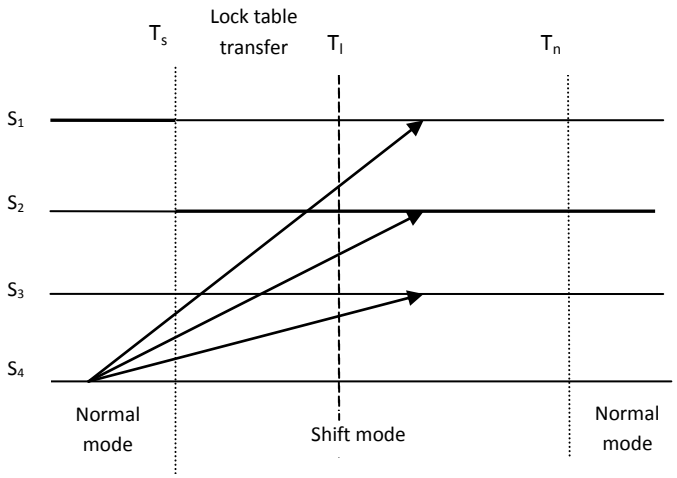

Fig. 1d: s_4 sends update notifications before shift mode starts and notifications arrived after the arrival of lock_table at s_2

Under this protocol we guarantee that no transactions/lock requests loss since the old primary continues to forward all locking requests and transactions to the new primary until it announces its leadership to all replicas after receiving the *lock_table*. Moreover, the shift mode duration, T_n-T_s, is too short compared to the election time. Our suggested protocol is defined by a set of rules that apply during the primary shift mode. These rules are described below:

Rule 1: is a modified version of rule 2 in the previous section that holds on the new transaction (no lock request is issued by the replica to serve the transaction during shift mode) and is executed at all replicas. This rule is shown below.

Rule 1:
$\boldsymbol{upon\ event} < trans(\ \ )|proc(\ \ ), O, M, C_i > and\ mode = shift\ do$
$create\ \ new\ tid$
$abort\_list = abort\_list \cup tid$
$client\ \boldsymbol{Notify}\ < tid, abort >$

Rule 2: is executed at all replicas in the system as follows:
1) Fire the shift mode at all replicas at the same time as a result of the occurrence of an outage in the outage table.
2) Identify the new primary for all replica
3) Trigger the old primary to transfer the *lock_table* to the new primary

Rule 2:
$\boldsymbol{upon\ event} < replica_{local\_time} = event\_time > do$
$mode = shift$
$new\_primary = get\_newprimary(schedul\_event)$
$tempt = primary$
$primary = new\_primary$
$if\ (temp = self\ )then$
$primary\ \boldsymbol{Trigger} < lock\_table()|lock\_table, self, T_{ptstamp}\ >$

Rule 3: is used by the new primary as follows:
1) Construct the *lock_table* delivered by the old primary
2) Reply to every lock request (i.e., it replies with either *ok_lock* or *ko_lock* messages as a response for any lock request and releases object locks as a response for update message)
3) Test if the *request_queue* contains pending requests, serve them as usual until the queue is empty (which means all postponed requests are served)
4) If the *lock_table* and the *request_queue* are empty, then notify all replicas to begin the normal mode which means that all normal rules shown in the previous section now apply.

Rule 3:
$\boldsymbol{upon\ event} < locktable()|lock\_table, S_i, T_{ptstamp}\ > do$
$lock\_table = lock\_table$
$lockdeleiverd = true$
$T_{ptstamp}\ = T_{ptstamp}$
$\forall\ request\ \epsilon\ request\_queue\ \boldsymbol{Trigger} < event()|self) >$
$request\_queue = request\_queue - request$
$if(\ lock\_table\ is\ empty\ )\ and\ (request\_queue\ is\ empty)$
$\forall\ S_i\ \in membership$
$\boldsymbol{Trigger} < ready()|self) >$
$mode = normal$

Rule 4: is a modified version of rule 3 in the previous section. It works in the shift mode as follows:
1) If the node is primary and the *lock_table* has been received, the primary replies by *ko_lock* if an object is already locked, otherwise it replies by *ok_lock* and updates the *lock_table*
2) If the node is primary and *the lock_table* has not been received, put the request in the *request_queue.*
3) If the node is not primary, forward the locking requests arrived to the old primary from any replica to the new primary. This prevents loss of transactions during primary shift.

Rule 4:
$\boldsymbol{upon\ event} < lock()|\ O, S_i, tid > and\ mode = shift\ do$
$\boldsymbol{if}\ primary$
$if(lockdeleiverd)$
$T_{ptstamp}\ = T_{ptstamp}\ + 1$
$flag\ = true$
$\forall\ \boldsymbol{o_i}\ \epsilon\ O\ \ if\ \sqsubseteq \boldsymbol{o_i}\ |\ \epsilon\ lock\_table\ then\ flag = false$
$if\ (flage = false)$
$\boldsymbol{S_i}$ Notify < $kolock()\ |\ tid, T_{ptstamp}\ >$;
$else$
$\forall \boldsymbol{o_i}\ \epsilon O\ \ lock\_table = lock\_table \cup \{< \boldsymbol{o_i}, \boldsymbol{S_i}, \boldsymbol{tid} >\}$
$\boldsymbol{S_i}$ Notify < $oklock()\ |\ tid, T_{ptstamp}$
else
$request\_quee = request\_queue \cup < lock()|\ O, S_i, tid >$
$\boldsymbol{else}$
$primary\ \boldsymbol{Trigger} < lock()|\ O, S_i\ tid >$

Rule 5: is a modified version of rule 5 in the previous section. It works in the shift mode as follows:

1) If the node is primary and the *lock_table* has been received, free the locked objects, execute update procedure, and notify client
2) If the node is primary and the *lock_table* has not been received, put the request in the *request_queue.*
3) If the *lock_table* and the *request_queue* are empty, then notify all replicas to begin the normal mode which means that all normal rules shown in the previous section now apply.
4) If the node is not primary, execute update procedure and notify client

Rule 5:
$\boldsymbol{upon\ event} < update()|\ proc(), S_i, tid, T_{ptstamp}\ > and\ mode = shift\ do$
$\boldsymbol{if}\ primary$
$\boldsymbol{if}(lockdelieverd)$
$T_{ptstamp}\ = T_{ptstamp}\ + 1$
$\forall oi\ \epsilon O\ lock\_table = lock\_table - \{oi, \_, \_\}$
$execute\ proc()$
$transcommit = transcommit\ \cup tid, T_{ptstamp}$
$client\ \boldsymbol{notify} < tid, update >$
$\boldsymbol{else}$
$request\_queue = request\_queue \cup$
$< update()|\ proc(), S_i, tid, , T_{ptstamp}\ >$
$if(\ lock\_table\ is\ empty\ )\ and\ request\_queue\ is\ empty$
$\forall\ \boldsymbol{S_i}\ \in membership$
$\boldsymbol{Trigger} < ready()|self) >$
$mode = normal$
$\boldsymbol{else}$
$execute\ proc()$
$commit_{list} = commit_{list} \cup tid\ \#\ T_{ptstamp}$
$client\ \boldsymbol{notify} < tid, update >$

Finally, when the new primary notifies all nodes in the systems by *ready()* message, shift mode is ended and nodes go to the normal mode again. In the normal mode, rules 1-5 are deactivated, hence, all nodes apply the PBR protocol rules again as shown in rule 6.

Rule 6: is executed at all replicas in the system

Rule 6:
upon event $<ready()|S_i>$ *do*
mode= normal

4. SIMULATION RESULTS

In order to test the feasibility of our approach, we built a prototype that represents both PBR approach and primary shift replication (PSR) approach. The prototype comprises of the application model and the system model. These models are described below:

System model: comprises of a set of servers (replicas) that are connected to each other and communicate via message passing through a TCP/IP network and a set of clients that issue transactions.
Application model: the application is simply represented by a counter which is initiated to zero and is incremented when a transaction is committed.

4.1 Experiments Setup

We have conducted a set of experiments that show the merits of our approach. The replication system comprises of four replicas (P_1, P_2, P_3, P_4), three clients with three shared objects (each client updates only one object). Each client issues 2000 transactions with a rate of one transaction every 150 msec which represents the network delay from client to server. The transmission delay between replicas is set to 25 msec. We made the servers faster than the clients to guarantee that all issued transactions are committed. Experiments were conducted to show how the primary role is transfer (shifted) in PSR. In order to test the stability of our approach, we conducted the PSR experiments at four different values of shift time, that is PSR1, PSR2,PSR3 and PSR4 which refer to values 1,2,3, and 4 seconds for the shift time respectively.

4.2 Experimental Results

In the experiments, we record the following measurements:

- Total number of committed transactions
- Total number of aborted transactions
- Number of messages (i.e., lock requests, *ok_lock*, *ko_lock*, and update notifications)

Based on these measurements, we studied the following relationships:

- The workload (i.e., total number of messages at each replica) distribution in PBR approach and in PSR
- Number of aborted transactions and the time duration of shift mode

Figure 2 shows the distribution of messages on replicas in both PBR and PSR. A quick inspection for this graph revealed the balanced distribution of message for all replicas in PSR compared to imbalanced distribution in PBR approach. It's obvious that in PBR approach that the primary (P_1) is heavily loaded by 18000 messages which represent 6000 lock requests, 6000 *ok_lock*, and 6000 update notifications, while other replicas are loaded only by 6000 messages which represent update notifications only since they do not receive any lock requests. On the other hand, in PSR each replica is loaded with approximately the same number of messages (9000), that is 6000 updates, around 1500 lock requests, and around 1500 *ok_lock* messages. This is because the primary role is distributed among all replicas.

Figures 3 and 4 illustrate the distribution of transactions on all replicas in both approaches. The figures show that in PBR approach all 6000 transactions (2000 transactions for each object) are served by the primary replica (P_1), while in PSR these transactions are distributed among all replicas (approximately 1500 transactions for each replica).

Figure 5 shows the number of aborted transactions as a function of the shift time. Intuitively, the number of aborted transactions increases as the shift time increases. This is due to the fact that no transactions are served during primary shift mode. Finally, it is worthy to mention that we have detected from 3 to 9 cases of forwarded lock requests from the old primary to the new primary or update notifications to the new primary inqueued since they arrived before the lock table is transferred. All of these cases are committed by the new primary.

In [15] Shirali *et. al.* computed the number of messages and the latency for a number of LEAs. In their results, the number of messages required to elect a new leader for 64 nodes was recorded from 128 to 176 messages. The latency ranges between from 3 to 128 time unit. In our system, we only need one message to transfer the *lock_table* regardless of the number of nodes in the system and the shift time is clearly much less than the recorded election time.

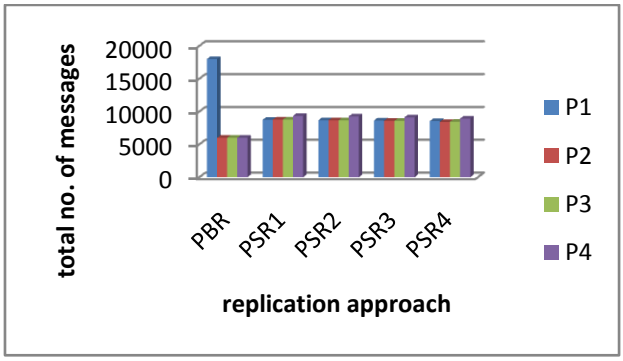

Fig. 2: Distribution of messages

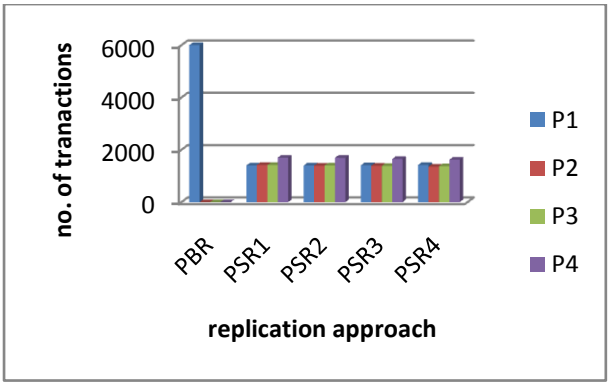

Fig. 3: Distribution of transactions

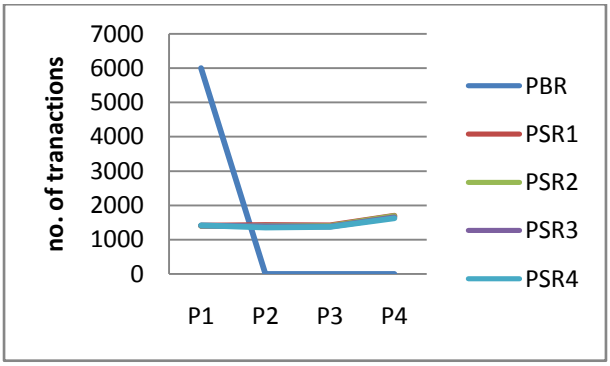

Fig. 4: Distribution of transactions

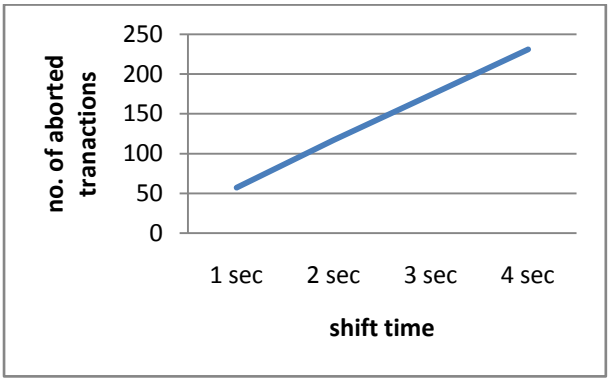

Fig. 5: No. of aborted transactions vs. transfer time

5. CONCLUSIONS

Traditionally, PBR systems employ Leader Election Algorithms (LEA) to recover from both planned and unplanned primary failures. During election, a replication system suffers from transaction loss, communication overhead and a notable delay time caused by the execution of LEA. In this paper, we introduced a novel protocol, called Primary Shift Replication (PSR), to manage the planned primary outages without using leader election protocol. The proposed approach shifts the primary role from the current primary to another scheduled node when a planned outage is about to occur. The no. of messages and communication time PSR is much less than the no. of messages and the election time in PBR, therefore, PSR improves systems availability. Moreover, PSR guarantees no transactions loss during the shift mode, hence, it preserves data consistency. In the future, we plan to apply the PSR on mobile environments.

6. REFERENCES

[1] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. 1993. The Primary-backup Approach. S. J. Mullender, editor, Distributed Systems, Addison-Wesley, (Chapter 8).

[2] Navin Budhiraja, Keith Marzullo. 1995. Tradeoffs in Implementing Primary-Backup Protocols. SPDP '95 Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing.

[3] Greg N. Frederickson and Nancy A. Lynch. 1987. Electing a Leader in a Synchronous Ring. J. ACM, 34(1):98–115.

[4] H. Garcia-Molina. Elections in a Distributed Computing System. 1982. IEEE Trans. Computers, 31(1):48–59.

[5] Suresh Singh and James F. Kurose. 1994. Electing Good Leaders. Journal of Parallel and Distributed Computing, Volume 21, Issue 2, pages 184–201.

[6] Scott D. Stoller. 2000. Leader election in asynchronous distributed systems. IEEE Transactions on Computers, 49(3):283–284.

[7]AL-Metwally Mostafa, Ilies Alouini. Fault Tolerant Global Store Module. (2004). http://www.mozart-oz.org/mogul/doc/metwally/globalstore/

[8] Valentin Mesaros , Raphaël Collet , Kevin Glynn , Peter Van Roy. 2005. A Transactional System for Structured Overlay Networks. Research Report RR2005-01, Universit´e catholique de Louvain, D´epartement INGI.

[9] E. Cecchet, G. Candea, and A. Ailamaki. 2008. Middleware-based Database Replication: The Gaps Between Theory and Practice. ACM SIGMOD Conference, Vancouver, Canada.

[10] Rachid Guerraoui and Luís Rodrigues. 2006. Introduction to Reliable Distributed Programming. Springer-Verlag Berlin Heidelberg.

[11] Y. Saito and M. Shapiro. 2002. Replication: Optimistic Approaches. Technical Report, Microsoft Research Ltd.

[12] W. Lang, J. Patel, and J. Naughton. 2009. On Energy Management Load Balancing and Replication. SIGMOD Record, Vol. 38, No, 4, 2009

[13] M. EffatParvar, N. Yazdani, Mehdi. EffatParvar, A. Dadlani, and A. Khonsari. 2010. Improved Algorithm for Leadership Election in Distributed Systems. 2nd International Conference on Computer Engineering and Technology.

[14] D.P. Gawali. 2012. Leader Election Problem in Distributed Algorithm. International Journal of Computer Science and Technology.

[15] Mina Shirali, Abolfazl HaghighatToroghi, and Mehdi Vojdani. 2008. Leader Election Algorithms: History and Novel schemes. Third 2008 International Conference on Convergence and Hybrid Information Technology.

[16] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design

& Implementation, Berkeley, CA, USA.

[17] Heutelbeck, D. and Hemmje, M. 2006. Distributed Leader Election in P2P Systems for Dynamic Sets. MDM, Page(s): 29 – 29.

[18] Zargarnataj, M. 2007. New Election Algorithm Based on Assistant in Distributed Systems", AICCSA 07, Page(s):324 – 331.

[19] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. 2001. http://aws.amazon.com/message/65648/