Make the Leader Work: Executive Deferred Update Replication

Maciej Kokociński Tadeusz Kobus Paweł T. Wojciechowski
Institute of Computing Science, Poznań University of Technology
Email: {maciej.kokocinski, tadeusz.kobus, pawel.t.wojciechowski}@cs.put.edu.pl

***Abstract*—In this paper we propose executive deferred update replication (EDUR), a novel algorithm for optimistic concurrency control in distributed transactional memory and database systems. EDUR streamlines transaction certification (i.e., checking for conflicts between concurrent transactions) with the broadcast protocol. which improves scalability and overall performance compared to deferred update replication based on total order broadcast (TOB). EDUR uses executive order broadcast (EOB), a novel protocol that can be seen as a generalization of TOB. Compared to TOB, EOB features new primitives and properties that enable the application to delegate some work to a leader—a process inherently present in many TOB algorithms that is responsible for coordination of message dissemination. The results of experimental evaluation show significant performance gains when using our approach.**

I. Introduction

Replication is typically used for building dependable and highly available services. It means deployment of a service on multiple machines and coordination of their actions so that a consistent state is maintained across all the service replicas. In this paper, we focus on *deferred update replication (DUR)* [1] which is a well known replication method used for optimistic concurrency control in distributed transactional memory [2] and database systems [3]. It requires every service replica (process) to maintain a full copy of all shared data items (objects). Transactions, which can be executed by any process, operate on local copies of shared objects. Only upon commit the processes synchronize so that they update their states in a consistent manner. The update is finalized only if a transaction is successfully *certified* (i.e., no conflicts with concurrent but already committed transactions have been detected). Otherwise the transaction is rolled back and restarted. DUR guarantees globally serializable execution of transactions.

Many authors advocate implementing DUR on top of the total order broadcast (TOB) (see e.g., [4][5]). Using TOB for disseminating messages limits the number of costly network communication steps and avoids the possibility of blocking. However, this approach is not free of weaknesses. Limiting the number of communication steps comes at the cost of performing transaction certification by each process independently. Moreover, the volume of data required to be exchanged between the processes can be large. Both factors limit DUR's scalability, especially with workloads that are characterized by long running transactions that access multiple shared objects.

In [6], we compared TOB-based DUR with state machine replication (SMR) and eventually merged the two approaches for better scalability and overall performance [2]. It turns out that using DUR on modern multicore hardware often gives worse results than executing the transactions sequentially on all replicas, as in SMR. The reason for this is the overhead caused by ensuring transactional semantics. In DUR this is especially evident for small, short-lived transactions. Furthermore, large, long-running transactions cause a lot of conflicts. It means that a high percentage of the transactions must be rolled back and restarted, thus diminishing the overall system performance. We sought for a way to make DUR more scalable.

In this paper we propose *executive deferred update replication (EDUR)*—a novel concurrency control algorithm that overcomes the limitations of TOB-based DUR. In EDUR transaction certification is streamlined with the broadcast protocol. Transactions are certified only by a leader, a designated process inherently present in some of TOB algorithms, responsible for coordination of message dissemination. EDUR requires the same number of communication steps as DUR but only one process performs transaction certification. Additionally, EDUR greatly reduces the network traffic compared to DUR, because the data needed for transaction certification is sent only to the leader. Moreover, a transaction's updates are broadcast to all processes only in case of successful certification.

However, as we show in this work, TOB is not sufficient for EDUR to run correctly. A protocol with stronger guarantees than TOB is necessary. For this purpose, we introduce *executive order broadcast (EOB)*, a novel protocol that can be perceived as a generalization of TOB. Compared to TOB, EOB features new primitives and properties necessary to capture the special rôle of the (changeable) leader process. The application can ask the leader to extract the semantic information from the messages and transform them according to the application's logic before sending the messages to other processes. We show that EOB can be efficiently used to implement EDUR.

To evaluate the performance of EDUR, we compare it to TOB-based DUR. For this purpose we extended Paxos STM, our DTM system featured earlier in [6][2], so that it can use two different transaction certification modules: one employing a classic DUR protocol built on top of TOB, and another employing EDUR running on top of EOB. The comparison is fair since in both cases we use JPaxos [7] as the basis for the implementation of the broadcast protocols. In all tests, EDUR achieves much better overall performance compared to DUR.

II. Related work

Due to limited space, we only discuss work most closely related to ours (see [6][2] for more references).

Postgres-R [3] is another TOB-based protocol for optimistic replication, originally proposed for database systems.

It was designed to solve the problem of high network traffic, which is known to limit the scalability of DUR. Compared to DUR, Postgres-R reduces the certification overhead and the volume of data that need to be exchanged between the processes at the price of a higher number of communication steps. EDUR further improves over Postgres-R in terms of certification overhead and network traffic. At the same time, it maintains the low number of communication steps as in DUR by streamlining certification with the broadcast protocol.

PO-broadcast [8] offers similar ordering guarantees as EOB. However, PO-broadcast is inherently view-based and is defined for primary-backup systems where only one process can broadcast update messages. Hence, EOB is more general than PO-broadcast, as it allows multiple processes to broadcast messages. A similar, view-based abstraction is extended virtual synchrony (EVS) [9]. EOB can be implemented with EVS. In fact EVS is strictly stronger than EOB, because it includes explicit group membership service. This feature of EVS has, however, a detrimental impact on availability of a replicated system (see §IV-C for more discussion).

Our implementation of TOB extends Paxos [10], a fast, reliable and well researched distributed agreement protocol. However, the EOB semantics can also be realized by extending any sequencer-based TOB protocol (see [11] for a survey).

III. System model

We assume an asynchronous system that consists of a set of processes. A process p_i is said to be *correct* if there is a time after which p_i always correctly executes its program. Otherwise p_i is *faulty*, i.e., it fails by *crashing* and never *recovers*, or crashes and recovers infinitely many times. No assumption is made on the relative speeds of the processes. To be able to solve distributed consensus, we assume the presence of a failure detector Ω. Each process has access to volatile memory and to stable storage (data stored in it survive crashes). Processes form a network by maintaining bidirectional *fair-loss links* between each pair of processes. They communicate only by exchanging messages (equipped with system-wide unique identifiers). Messages may be lost or dropped by the links and no upper bound on message transmission is known. The failure pattern of links is independent of the one of the processes.

IV. Problem statement

The aim of this section is to show that EDUR requires a coordination mechanism stronger than total order broadcast (TOB) but weaker than some view-based solutions such as extended virtual synchrony (EVS). We assume all the discussed broadcast protocols to be uniform.

A. The idea behind executive deferred update replication

Consider the TOB-based DUR protocol, discussed in [1]. In this replication scheme, before a transaction can be committed, the processes need to synchronize. A process that executed a transaction broadcasts a message which contains the updates of the transaction and metadata necessary for certification. Then, each process independently certifies the transaction against the concurrent, but already committed transactions. A transaction is committed only if certification is successful. Otherwise the transaction is rolled back and restarted. All processes change their states in the same way since the certification procedure is deterministic and each process receives messages in the same order. TOB-based DUR limits the number of costly communication steps because it requires only one broadcast per transaction's run. However, minimizing the number of communication steps comes at the price of increased utilization of resources. Firstly, certification has to be performed by each process independently. Secondly, the volume of data exchanged via network is high, mainly due to, usually large, metadata that are broadcast alongside updates. Thus, DUR scalability is limited, especially for long running transactions that access multiple shared objects.

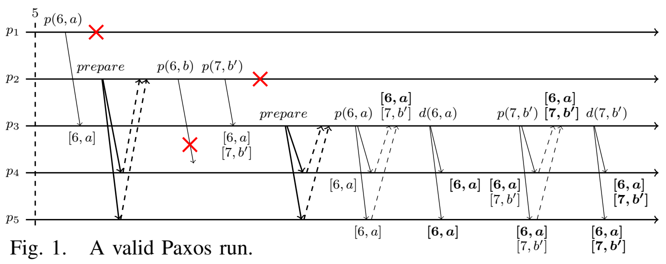
Many of the existing TOB algorithms are *sequencer-based* (see [11] for a survey). It means that there is a designated process called *the sequencer* (or *the leader*) which coordinates message exchange between all processes. The leader is responsible for establishing the final order of message delivery by, essentially, stamping each message with a sequence number. The key idea behind EDUR is to employ the leader to certify transactions on behalf of other processes. To this end, the leader processes and modifies the messages it receives before sending them to the rest of processes. Upon receipt of a message, the leader extracts the transaction metadata and uses them to certify the transaction. Then, if certification is successful, the leader forwards only transaction's updates, as the rest of the processes do not need to certify the transaction again. On the other hand, if the transaction fails certification no data need to be broadcast. The leader only sends an information to the process that originally executed the transaction that the transaction has to be re-executed.

Streamlining transaction certification with the broadcast protocol has several advantages. Firstly, the total amount of computation is reduced (transaction is certified only by a single process). Secondly, once a transaction is certified, the metadata which are often large, do not have to be forwarded to the rest of the processes. The amount of data needed to be sent via network is further reduced if a transaction fails certification. Note that the load of the leader does not increase: both in DUR and EDUR the leader has to certify transactions. In EDUR certification happens earlier, before the leader establishes the final order of message delivery and forwards the message to the rest of the processes. EDUR requires the same number of communication steps as DUR.

B. Anomalies in total order broadcast

In order to build EDUR one cannot just use a naive extension of a sequencer-based TOB protocol that exposes the leader to the application. The reason for this is twofold. Firstly, the leader, by performing some additional work before sending messages to the rest of the processes, imposes an implicit dependency on the final message delivery order. Potentially each message delivered to an application depends on the one delivered previously. Secondly, to achieve high performance, TOB protocols concurrently execute several rounds of the protocol (called *instances*), each coordinating the exchange of a batch of messages. As we demonstrate below, an erroneous behaviour can occur when the leader changes.

In Figure 1 there is a valid execution of Paxos for five processes $p_1, ..., p_5$. Each of the processes maintains a sequence



p_1
p_2
p_3
p_4
p_5
5
$p(6,a)$
prepare
$p(6,b)$
$p(7,b')$
$[6,a]$
$[6,a]$ $[7,b']$
prepare
$p(6,a)$
$[\mathbf{6},\boldsymbol{a}]$ $[7,b']$
$d(6,a)$
$p(7,b')$
$[\mathbf{6},\boldsymbol{a}]$ $[\mathbf{7},\boldsymbol{b'}]$
$d(7,b')$
$[6,a]$
$[\mathbf{6},\boldsymbol{a}]$
$[\mathbf{6},\boldsymbol{a}]$ $[7,b']$
$[\mathbf{6},\boldsymbol{a}]$ $[\mathbf{7},\boldsymbol{b'}]$

Fig. 1. A valid Paxos run.

where it records messages to be delivered to the application. The value v and the position in the sequence i (often called *the instance number*) for a given request is determined by the leader (denoted $[i, v]$). A value is locked for a given instance once processes reach an agreement and let the leader know about it (a locked value v for a given instance i is denoted $[\boldsymbol{i}, \boldsymbol{v}]$). In Paxos, reaching agreement requires the majority (quorum) of processes to vote in the same way. Concurrent execution of several instances means that a new instance can be started before the previous one is finished (a value is locked).

In our example, p_1 has been the sole leader up to instance 5, i.e., p_1 succeeded in reaching agreement on all requests it proposed in instances 1 through 5. Process p_1 proposes value a for instance 6 (sends message $p(6, a)$) but succeeds in sending the proposition only to p_3 and then crashes (therefore not locking a for instance 6). Process p_2 suspects that p_1 has crashed so p_2 executes the prepare phase of Paxos in order to become the leader. Upon gaining quorum of votes (p_2 is recognized to be the most recent leader process by the majority of processes) p_2 concurrently tries to propose values b and b' for instances 6 and 7. However, p_2 succeeds only in sending a message with b' to p_3. After p_2 crashes, process p_3 executes the prepare phase in order to become the leader. However, for the already initiated instances 6 and 7 process p_3 cannot propose just any values. To guarantee safety in Paxos, once p_3 knows that some other processes (previous leaders) proposed some values for the given instances, it has to use them first for those instances. Process p_3 chooses then a for instance 6 and b' for instance 7. Eventually, those values are adopted by the majority of processes, i.e., they agree to deliver messages a and b' for instances 6 and 7. This behaviour poses no problem if the messages broadcast using TOB are self-contained and do not depend on the messages previously agreed upon. Otherwise, the message order established by Paxos (or any other TOB protocol) may not be correct from the application's point of view. Evidently, locking b' for instance 7 does not make any sense if b' is logically dependent on b.

Imagine that the messages above contain the modifications that are the effects of transaction execution. EDUR must ensure that the dependency order imposed by the leader is always maintained. Clearly, TOB cannot be used to implement EDUR. Note that *causal order broadcast* [11] is not suitable too, as it only establishes precedence between the messages broadcast and delivered by processes. It fails to reflect the precedence imposed by the streamlined certification.

C. View-based broadcast solutions

EDUR can be built on top of view-based group communication protocols (see e.g., EVS [9]). This way, the execution is organized in a sequence of views, each giving processes a perception of the system as consisting of only correct processes. Group membership service is responsible for triggering a view change every time there is a process suspected to have crashed or that wants to join or leave the group. Moreover, the broadcast protocol guarantees a total order of message delivery that respects the order of views. It is therefore safe to explicitly elect a leader in every view. In EVS-based EDUR, upon commit each process sends a unicast message to the current leader which in turn certifies the transaction and forwards the updates to the rest of the processes.

However, view-based solutions are not optimal for EDUR. EDUR does not require a full group membership service. Introducing view-based computation simplifies the recovery process, however, recovery and fault-tolerance can be achieved just as efficiently without it. In a large network false suspicions regarding process crashes can be frequent, thus causing unnecessary reconfigurations. Moreover, the computation has to be paused during installation of a new view. To avoid such issues the crash of any process other than the leader should be transparent, similarly as in passive replication. In order to overcome the limitations of both TOB and EVS we propose a new broadcast protocol called executive order broadcast.

V. Executive Order Broadcast

In this section, we formalize the primitives and properties of executive order broadcast (EOB). Our definition builds on the specification of total order broadcast (TOB), as in [12].

A. The EOB primitives

The core primitives of EOB are similar to those from TOB:

- *EO-Broadcast(m)* broadcasts a message m to all processes,
- *EO-Deliver(m)* delivers a message m broadcast by the process $sender(m)$.

Additionally four new primitives are introduced that capture the special role of the leader process:

- *EO-LeaderElect(H_I)* informs a process, that it becomes a leader; H_I is a possibly empty sequence of messages soon to be delivered which constitute the initial knowledge of the leader,
- *EO-LeaderRecall* informs a process, that it is no longer the leader,
- *EO-LeaderDeliver(m)* delivers message m to the leader process, so m can be processed,
- *EO-LeaderBroadcast(m)* is used by the leader process to forward a transformed message to all processes.

The EO-LeaderElect and EO-LeaderRecall primitives are used by a local failure detector to inform the process when it has to take on the duties of the leader. The EO-LeaderDeliver and EO-LeaderBroadcast primitives express the streamlined application logic intended to be performed by the leader. To emphasize that a message sent by a process can change before it is EO-Delivered by other processes we denote a message m as a tuple $m = (id, mc)$ where id is a system-wide unique message identifier and mc is the message content which can be modified .[1]

[1]It implies that if a process p_i executes EO-Broadcast $m = (id, mc)$ no other process p_j ever executes EO-Broadcast $m' = (id, mc')$.

A typical sequence of actions is as follows. First, a process p_i EO-Broadcasts a message $m = (id, mc)$. The message is first forwarded to the current leader process p_k (could be the same process) and then p_k EO-LeaderDelivers m. The leader transforms the message m to $m' = (id, mc')$ according to the application's logic using its unique knowledge about the messages EO-Broadcast but not yet EO-Delivered by other processes. The leader then EO-LeaderBroadcasts m'. Finally m' is EO-Delivered by all correct processes.

B. The complimentary definitions

Below we inroduce two auxiliary terms: a reign period and message precedence:

Definition 1. *The* ***reign period*** *r of a process p_i begins when p_i executes the EO-LeaderElect(H_I) event e for some message history H_I and lasts until either p_i executes the first EO-LeaderRecall event after event e, or until p_i crashes. If p_i neither crashes nor executes EO-LeaderRecall then r is unbounded. H_I is the* ***initial history*** *of r and is denoted by $initialHistory(r)$.*

Although the specification of EOB allows for multiple concurrent leaders, reign period gives a process an impression of being a sole leader, capable of making authoritative decisions on behalf of other processes. Before a process can take on the role of a leader, it has to be up-to-date, as explained in §IV. Otherwise the decisions made and EO-LeaderBroadcast by this process must be considered by other processes as not valid and can never be EO-Delivered. Becoming up-to-date means making sure that all messages successfully forwarded by the previous leader are EO-Delivered by the new leader, which obviously can take some time. Providing the new leader with the initial history allows the leader to immediately assume its responsibilities. This optimization is best effort–sometimes the given initial history can turn out to be incomplete or incorrect. However, as we show later in this section, no inconsistency can occur; in the worst-case scenario some messages EO-LeaderBroadcast by the leader will have to be discarded upon being received by the processes.

The order of message delivery imposed by the leader is reflected by the notion of logical precedence of messages:

Definition 2. *Let $m = (id, mc)$ be a message EO-LeaderBroadcast by some process p_i in its reign period r, such that $initialHistory(r) = H_I$. A message $m_p = (id_p, mc_p)$ is the* ***precedent message*** *of the message m, if:*

1) *m_p is the last message EO-LeaderBroadcast by p_i in r before m, or if there is no such message,*
2) *m_p is the last message from H_I, or if $H_I = \emptyset$,*
3) *m_p is the last message EO-Delivered by p_i before r began, or if there is no such message,*
4) *$m_p = m_\perp = (\perp, \perp)$.*

We use $m \prec m'$ to denote that message m precedes m'.

C. The EOB specification

Definition 3. *The properties of executive order broadcast are:*

Leader reign: *A process p_i in its reign period r such that $initialHistory(r) = H_I$ only EO-Delivers messages from H_I and those EO-LeaderBroadcast by p_i in r;*

Termination: *If a process p_i EO-Broadcasts a message $m = (id, mc)$ and then p_i does not crash, then p_i eventually EO-Delivers $m' = (id, mc')$;*

Validity: *For any message $m = (id, mc)$:*

1) *every process p_i that EO-Delivers m, EO-Delivers m only if m was previously EO-LeaderBroadcast by some process p_j, and*
2) *every process EO-Delivers $m^* = (id, *)^2$ at most once;*

Leader validity: *For any message $m = (id, mc)$:*

1) *a process p_i EO-LeaderDelivers m only if m was previously EO-Broadcast by some process p_j, and*
2) *a process p_i in its reign period r EO-LeaderBroadcasts $m' = (id, mc')$ only if m was previously EO-LeaderDelivered by p_i in r, and*
3) *a process p_i EO-LeaderDelivers m and EO-LeaderBroadcasts $m^* = (id, *)$ at most once within a given reign period of p_i;*

Agreement: *If a process EO-Delivers a message m, then every correct process eventually EO-Delivers m;*

Executive order: *For any message $m = (id, mc)$ EO-LeaderBroadcast by a process p_k in its reign period r, a process p_i EO-Delivers m only if:*

1) *p_i did not EO-Deliver any other message so far and $m_\perp \prec m$, or*
2) *the last message p_i EO-Delivered before m is the precedent message of m.*

The Agreement and Validity properties were adopted from TOB. They state that each correct process EO-Delivers the same set of messages. Additionally, the EO-Delivered messages are not created out of thin air and cannot be EO-Delivered more than once. Termination is nearly identical to its equivalent in TOB. The only difference lies in accounting for a message being transformed from $m = (id, mc)$ to $m' = (id, mc')$. Without Termination property any EOB algorithm could trivially satisfy the rest of properties by not exchanging any messages. Our specification lacks the Total Order property present in TOB. In EOB the Total Order property is replaced by a new stronger property called Executive Order. Executive Order ensures that each process EO-Delivers messages according to the order imposed by the leader. The messages form a sequence in which each message (except the first one) is EO-Delivered only after the precedent message is EO-Delivered. Additionally we specify two properties, necessary to ensure correctness of algorithms such as EDUR. Firstly, the Leader Reign property guarantees that a leader ends its reign period once it detects that some other process, also acting as a leader, successfully forwarded some messages. It implies that, before becoming the leader again, a process has to know about all the actions undertaken by the previous leader. Secondly, Leader Validity restricts the actions of a leader, so it does not process the same message multiple times, and it does not create messages out of thin air.

Note that progress is not guaranteed by the above specification of EOB. For instance, consider an EOB-based application that drops some messages that it EO-LeaderDelivers. Then,

$^2(id, *)$ represents a message with a unique id and any message content.

all properties except Termination are guaranteed. Thus, some additional requirements on the application are necessary, since the application's logic is part of the EOB broadcast protocol. Below we give the Leader Termination property that requires the application to eventually execute EO-LeaderBroadcast $m' = (id, mc')$ after EO-LeaderDelivering $m = (id, mc)$ in a given reign period.

Definition 4. ***Leader termination****: If a process p_i in its reign period r EO-LeaderDelivers $m = (id, mc)$ and r is unbounded, then p_i eventually EO-LeaderBroadcasts $m' = (id, mc')$.*

D. The characteristic of EOB

Under stable conditions, i.e., when a leader does not change, EOB behaves similarly to a sequencer-based TOB: all messages pass through the leader, the leader transforms them according to the application logic, and finally they are EO-Delivered to all processes in the order established by the leader. Let us reconsider the example from §IV. The process p_3 was successful in locking values a and b' for instances 6 and 7 (a and b' were originally proposed by p_1 and p_2, respectively), although b' is logically dependent on b (unsucessfully proposed by p_2). In EOB such scenario is not possible: b would be the precedent message of b', thus b' could not be EO-Delivered to the application unless b is EO-Delivered first.

Note that, in EOB changes of the leader occur smoothly. The initial history allows the new leader to immediately assume its duties, thus the leader transition period is much shorter compared to the view-based solutions, where the leader first has to be elected and then gather the information about all previously started instances. Moreover, the performance of EOB in case of failures is at worst the same as in TOB, and can be even better (when some messages are dropped due to missing precedent ones).

VI. EXECUTIVE DEFERRED UPDATE REPLICATION

The pseudocode for EDUR is given in Algorithm 1. The transaction's execution phase proceeds in the same way as in TOB-based DUR. Transactions, which consist of a series of read and write operations, are executed concurrently by different processes (or threads) on the copies of shared objects. Each transaction has a unique id and maintains two sets: $updates$ and $metadata$. The former set is used to store the copies of the shared objects modified by it. The latter set maintains the information necessary for transaction certification, i.e., objects read and written by the transaction. Once COMMIT is called, marking the transaction's transition from the executing to committing state, the transaction's id with $updates$ and $metadata$ are EO-Broadcast. The message is received by the EOB leader process which then EO-LeaderDelivers it. Now, the leader certifies the transaction on behalf of other processes. The leader uses the metadata to check whether the transaction has read some stale objects (i.e., modified by the concurrent but already committed transactions or successfully certified by the leader in its current reign period). If so, the transaction has to be aborted. On the other hand, if the certification is successful, each process has to update its state according to the changes performed by the transaction. The leader transforms the message depending on the certification outcome and then EO-LeaderBroadcasts it. If the certification was successful, the transformed message contains the transaction id and the $updates$ set, so each process that EO-Delivers the message, can update its state. Otherwise only id is broadcast. Upon receipt of such a message, the process that executed the transaction rolls back and restarts it.

It is easy to see why EDUR works during stable periods, (when a leader process does not change). All messages pass through the leader which certifies, transforms and finally forwards them to all processes. It means that an implicit order on message delivery is introduced. The leader does not wait for a transaction it successfully certified to be committed before it certifies other transactions. Since the leader does not change, each process EO-Delivers messages in the order the leader sent them. The consistency is therefore preserved. During unstable periods the consistency is preserved as well. The order in which the leader EO-LeaderBroadcasts messages corresponds to the relation of precedence between the certified transactions. EOB guarantees that the precedence order is always respected, thus no inconsistencies can arise. The performance under failures is comparable to DUR's since changes of the leader occur smoothly (as described in section V-D).

EDUR improves upon TOB-based DUR in several ways. Firstly, certification of each transaction is performed only once, not by every replica as in DUR. EDUR does not increase the load of the leader compared to DUR, where each process (including the TOB leader) has to certify transactions either way. Secondly, EDUR greatly reduces network traffic. It is because the $metadata$ set, which in DUR is broadcast to all replicas, in EDUR is sent only to the leader. Thirdly, in case a transaction fails certification, only its id has to be broadcast, thus saving on sending both the $updates$ and $metadata$ sets. In total, then, compared to DUR, EDUR reduces the overall cost of performing certification by the factor of n and the volume of exchanged data by at least $(n-1) \times size(metadata)$. This allows EDUR to scale much better than DUR (as evidenced in §VII). Several futher optimizations are possible that are not applicable to DUR, which can further improve scalability and lower the burden of the leader process.

VII. EVALUATION OF EDUR

We compare the performance of EDUR and TOB-based DUR using the hashtable microbenchmark. It features a simple hashtable that can be managed through three operations: *get*, *put* and *remove*. The hashtable stores a fixed number of key-value integer elements from a defined range. A single run of the microbenchmark consists of a series of requests issued to the hashtable. Each request is either an RO or RW transaction. An RO transaction performs 100 *get* operations with a randomly chosen set of keys. An RW transaction also performs 100 operations but two of them modify the hashtable. The decision whether to insert a new element to the hashtable or remove an existing one depends on results of the earlier *get* operations. This way the original saturation of the hashtable that is set to 50% can be maintained. Our test can be performed with a varying mix of RO and RW transactions. Due to lack of space, we only present one of the scenarios, namely 50/50. We run this benchmark on a cluster of 10 nodes with two setups. In the first one (called *Hashtable: high contention*) the hashtable's size is set to 10000. In the second one (called *Hashtable:*

Algorithm 1 The executive deferred update replication algorithm for process p_i

```
1:  $committedTx \leftarrow \emptyset$; $processedTx \leftarrow \emptyset$
2:  function COMMIT(transaction $t$)
3:      EO-BROADCAST $(t.id, t.updates, t.metadata)$
4:  upon EO-LEADERDELIVER($m = (id, updates, metadata)$)
5:      $result \leftarrow$ CERTIFY($m$)
6:      if $result = success$ then
7:          $processedTx \leftarrow processedTx \cup \{(id, updates)\}$
8:          $m' \leftarrow (id, updates)$
9:      else
10:         $m' \leftarrow (id, \perp)$
11:     EO-LEADERBROADCAST $m'$
12: function CERTIFY($m$)
13:     check $m$ for conflicts $\forall t \in committedTx \cup processedTx$
14:     return { $success, failure$ }
15: upon EO-DELIVER($m = (id, updates)$)
16:     if $updates \neq \perp$ then
17:         $committedTx \leftarrow committedTx \cup \{(id, updates)\}$
18:         if $p_i$ is leader then
19:             $processedTx \leftarrow \{(id', u) \in processedTx : id' \neq id\}$
20:         atomically apply $updates$
21:     else
22:         if transaction with $id$ executed locally then
23:             restart transaction with $id$
24: upon EO-LEADERELECT($initialHistory$)
25:     for all $(m = (id, updates)) \in initialHistory : updates \neq \perp$ do
26:         $processedTx \leftarrow processedTx \cup \{(id, updates)\}$
27: upon EO-LEADERRECALL
28:     $processedTx \leftarrow \emptyset$
```

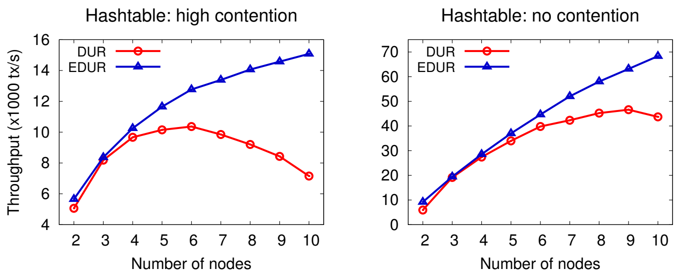


Fig. 2. Throughput for the Hashtable microbenchmark.

no contention), the hashtable's size is increased to 100000. Additionally, the key range is partitioned between the worker threads to eliminate conflicts between transactions altogether.

The evaluation results for both the high contention and no contention setups are given in Figure 2. In the the *high contention* setup the abort rate (i.e., the ratio between the number of aborted transactions to all transaction runs) ranges from 28% for 2 nodes to 69% for 10 nodes. DUR achieves the highest performance for 6 nodes and then the throughput diminishes. This can be attributed to the increased number of aborted transactions. The more nodes take part in the computation, the more concurrently executed transactions fall into conflicts and have to be rolled back and restarted. For 6 nodes, the output traffic from the leader exceeds 480 Mb/s, i.e., nearly the half of the maximum network throughput (1 Gb/s). For 10 nodes, the network is nearly saturated. On the other hand, EDUR continues to scale with the increasing number of nodes. It is because EDUR greatly reduces the network traffic. For 6 nodes the output traffic from the leader is about 48 Mb/s and raises to about 96 Mb/s for 10 nodes–10 times less than in case of DUR. Additionally, EDUR is able to greatly reduce the size of messages it broadcasts (from 550 B to 150 B).

Obviously, in the *no contention* setup EDUR cannot benefit from dropping updates of aborted transactions. However, this setup well demonstrates the benefits that result from performing the certification only on one node and the fact that metadata need not to be broadcast to all the processes. For these very reasons EDUR scales nearly linearly all the way up to 10 nodes. For 10 nodes EDUR uses only about half of the network bandwidth. On the other hand, the network is fully saturated for 9 nodes when using DUR. Adding more nodes would only diminish DUR's performance.

VIII. Conclusions

In this paper we presented executive deferred update replication, a novel optimistic concurrency control algorithm for replication in distributed transactional systems. EDUR uses executive order broadcast, a novel protocol which we specified in this paper alongside EDUR. EOB integrates closely with EDUR and allows for streamlining transaction certification with message broadcast. As we demonstrated, EDUR provides much better scalability and overall throughput than Deferred Update Replication based on Total Order Broadcast.

Acknowledgment This work was funded in part by National Science Centre grant 2011/01/N/ST6/06762.

References

[1] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication: Theory and Practice*, ser. LNCS, vol. 5959. Springer, 2010.
[2] T. Kobus, M. Kokociński, and P. T. Wojciechowski, “Hybrid Replication: State-Machine-based and Deferred-Update Replication Schemes Combined,” in *Proc. of ICDCS '13*, Jul. 2013.
[3] B. Kemme and G. Alonso, “Don't be lazy, be consistent: Postgres-R, a new way to implement database replication,” in *Proc. VLDB '00*, 2000.
[4] F. Pedone, R. Guerraoui, and A. Schiper, “Exploiting atomic broadcast in replicated databases,” in *Proc. of Euro-Par '98*, Sep. 1998.
[5] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, “Exploiting atomic broadcast in replicated databases,” in *Proc. of Euro-Par '97*, Aug. 1997.
[6] P. T. Wojciechowski, T. Kobus, and M. Kokociński, “Model-driven comparison of state-machine-based and deferred-update replication schemes,” in *Proc. of SRDS '12*, Oct. 2012.
[7] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, “JPaxos: State machine replication based on the Paxos protocol,” Faculté I&C, EPFL, Tech. Rep. 167765, Jul. 2011.
[8] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *Proc. of DSN'11*, 2011.
[9] M. Amir, L. E. Moser, Y. Amir, P. M. Melliar-smith, and D. A. Agarwal, “Extended virtual synchrony,” in *Proc. of ICDCS'94*, 1994.
[10] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, May 1998.
[11] X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Comput. Surv.*, vol. 36, no. 4, Dec. 2004.
[12] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, “Deconstructing Paxos,” *SIGACT News*, vol. 34, no. 1, 2003.