The Correctness Criterion for Deferred Update Replication

Tadeusz Kobus, Maciej Kokociński, Paweł T. Wojciechowski
Institute of Computing Science,
Poznan University of Technology, Poland
{Tadeusz.Kobus,Maciej.Kokocinski,Pawel.T.Wojciechowski}@cs.put.edu.pl

Abstract

In this paper we propose *update-real-time opacity*, a new correctness criterion based on opacity, that precisely describes the guarantees offered by Deferred Update Replication (DUR) protocol. We specify update-real-time opacity as a member of the ⋄ *opacity* family of properties, which we also introduce in this paper. We provide additional properties (as part of the ⋄ opacity family), which relax to various extent the transaction order requirements of opacity, in order to embrace a wider class of strongly consistent transactional systems. In the paper we discuss the relation between the members of ⋄ opacity and other popular correctness criteria used in the context of transactional systems.

Keywords correctness, deferred update replication, opacity

1. Introduction

Replication is an established technique used for building dependable and highly available services. In a replicated system, a service is deployed on multiple machines whose actions are coordinated, so that a consistent state is maintained across all the service replicas (processes). In this way the clients, that can issue requests to any of the replicas, are served despite of (partial) system failures.

Deferred Update Replication (DUR) [5] is one of the most widely employed protocols for concurrency control in database and distributed transactional memory (DTM) systems that use replication for achieving high availability.[1] In DUR, every request sent by a client to any of the replicas, is executed by the replica that received it as an atomic transaction, and then, the resulting updates are broadcast to all processes, so they can update their state accordingly. However, upon receipt of a message with a transaction's update, the processes do not update their state right away. In order to ensure that consistency is preserved across the system, all processes (independently) execute a certification procedure that checks if the transaction read any stale data. If so it has to be rolled back and restarted. Since all updates are delivered in the same order (by using, e.g., a Total Order Broadcast protocol [7]), all processes change their state in the same way. Since read-only transactions do not modify the system's state in any way, they do not require a distributed certification. Instead, only the process that executed a read-only transaction, certifies it to ensure that it has not read any stale data.

Opacity [9] [10] is often a desired property for transactional systems. Systems that satisfy opacity guarantee that no transaction (no matter whether live, aborted or committed) ever reads stale or inconsistent data. DUR, in fact, guarantees that no transaction ever reads inconsistent data, but allows live, aborted and read-only transactions to observe stale data.

To better understand why DUR breaks opacity, consider a system consisting of a few replicas (servers), where one of them is lagging behind. Client c_1 interacts with an up-to-date replica, while client c_2 interacts with the lagging one. Now, suppose that c_1 executes an update request (an updating transaction) and receives feedback from its replica. This update does not reach c_2's replica because of the lag. If the two clients communicate with each other, c_2 may notice it is missing an update, or even worse, it may not notice, but still take actions which depend on the update. Indeed, if c_2 starts a new transaction on its replica, contrary to its expectations, it will not be able to observe the update. If the transaction undergoes certification it will be aborted, but still up to some point in time (possibly until commit) it will execute on a stale snapshot.[2] Moreover, if the transaction executed by c_2 is a query (a read-only transaction), the transaction may even commit (no inter-process synchronization is required for read-only transactions). This clearly stands in contrast with the *real-time order* property guaranteed by opacity, i.e. if one transaction precedes another, all its effects must be visible to the latter. This requirement of opacity is usually difficult and costly to ensure in a distributed environment. The question is, then, which property can we use to properly describe what DUR actually guarantees?

Naturally, DUR guarantees serializability [18], a much weaker property compared to opacity. Serializability only requires that the execution of transactions is equivalent to some serial execution of all committed transactions. It does not impose any limitations on the order in which the transactions are serialized. In particular, in a serializable execution it is possible for two transactions executed sequentially by the same process to appear as if they were executed in a different order. Moreover serializability does not provide any safety guarantees to live or aborted transactions.

In this paper we propose *update-real-time opacity*, a correctness criterion based on opacity, which precisely describes the guarantees offered by DUR. We present update-real-time opacity as a member of the ⋄ *opacity* family of properties, that we also introduce in this paper. By providing a set of closely related properties that relax transaction order requirements of opacity to various extent, we are

[1] In this paper, we assume DUR based on Total Order Broadcast, as in [5]. However, our results do not depend on the way DUR is implemented. For a broader discussion on various implementations of DUR, see Section 2.

[2] To prevent live transactions from reading stale data, the system would have to run a distributed consensus round before the start of each transaction.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

TRANSACT '15, July 15–16, 2015, Portland, Oregon, USA.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

able to embrace a wider class of strongly consistent transactional systems. We show the relation between the properties and discuss how different order requirements impact the perception of the system's behaviour from the clients' point of view. We also prove that DUR relying on TOB satisfies update-real-time opacity.

The rest of the paper is structured as follows. In Section 2 we discuss the work relevant most closely to ours. Then, in Section 3 we provide the formal definition of the $\diamond$ opacity family of properties. Next, in Section 4 we briefly describe DUR and show why it guarantees update-real-time opacity. Finally, we conclude in Section 5.

2. Related Work

DUR, which we explain in more detail in Section 4, is the most basic protocol for achieving multi-primary-backup replication [5]. Various flavours of DUR are implemented in several commercial database systems, including Ingress, MySQL Cluster and Oracle. These implementations use 2PC [1] as the atomic commitment protocol. In this paper, we consider DUR based on Total Order Broadcast (TOB) [7]. This approach is advocated by several authors because of its nonblocking nature and predictable behaviour (see [2, 19, 20] among others). Most recently, it has been implemented in D2STM [6] and Paxos STM [23] [15]. It has also been used as part of the coherence protocols of S-DUR [22] and RAM-DUR [21].

There are a number of optimistic replication protocols that have their roots in DUR, e.g., Postgres-R [14] and Executive DUR [17]. The differences between these systems lie not in the general approach to processes synchronization, but in the way the transaction certification is handled. Both of these protocols, similarly to DUR, certify read-only transactions without inter-process synchronizations. It means that they are not opaque. However, one can show that they guarantee update-real-time opacity, similarly to DUR.

Over the years a multitude of correctness criteria have been defined for strongly consistent transactional systems. Serializability is the most basic of them all [18]. It specifies that all committed transactions are executed as if they were executed sequentially by a single process. Strict serializability [18] additionally requires that the real-time order of transaction execution is respected (i.e. the execution order of non-overlapping committed transactions is preserved). Update serializability [12] is very similar to serializability, but allows read-only transactions to observe different (but still legal) histories of the already committed transactions.

All three correctness criteria mentioned above regard only committed transactions and say nothing about live or aborted transactions. As briefly discussed earlier, sometimes this is not enough. Therefore, new correctness criteria emerged that formalize the behaviour of all transactions in the system, including live transactions. Although some of them, such as recoverability, avoiding cascading aborts or strictness [1] specify the behaviour of read and write for both live and completed transactions, but they say nothing about global ordering of transactions (unlike serializability and properties similar to it). This, in turn, limits their usefulness in the context of strongly consistent transactional systems. Therefore, our attention focuses on properties that maintain (in most cases) a global serialization for all transactions.[3]

The following properties maintain a global serialization only for some of the transactions. Extended update serializabilty [12] ensures update serializability for both committed and live transactions. Therefore, it features a global serialization for all the updating transactions (read-only transactions may observe a different serialization). Virtual world consistency [13] allows an aborted transaction to observe a different (but still legal) history.

Similarly as extended update serializabilty extends update serializability, opacity [9] [10] extends strict serializability to guarantee live transactions to always read a consistent state. It features a global serialization of all transactions. Rigorousness [4], TMS2 [8] and DU-opacity [3] offer even stronger guarantees. They restrict some particular sets of histories compared to opacity: rigorousness and TMS2 impose stronger requirements on the ordering of concurrent transactions, while DU-opacity explicitly requires that no read operation ever reads from a transaction that is not commit-pending or committed. Moreover, all these three properties are defined only in a model that assumes read-write registers. TMS1 [8] was proposed to slightly relax opacity. It allows not only each transaction, but even each operation, to observe a different view of past transactions. The possible histories are, however, restricted by a few conditions, which enforce quite strong consistency (despite lack of a global serialization). All of the properties mentioned above, including opacity, require that the real-time order of transaction execution is respected, similarly as in case of strict-serializability.

Update-real-time opacity proposed in this paper can be seen as a blend of various features of the above properties. On one hand it resembles (extended) update serializability, because it differentiates between updating and read-only transactions. On the other hand, it guarantees that all transactions (regardless of their type or current state of execution), share a common equivalent history of transactions' execution, as in opacity. However, in update-real-time opacity only committed updating transactions need to respect real-time order of transaction execution. Live and read-only transactions can operate on a stale snapshot of shared objects.

The $\diamond$ opacity family of properties currently features six members, ordered by the strength of offered guarantees. The strongest property of them all is *real-time opacity* which is equivalent to opacity defined in [10]. *Commit-real-time opacity* allows live and aborted transactions to read stale (but still consistent) data. *Write-real-time opacity* futher relaxes real-time order guarantees on transactions that are known *a priori* to be read-only. *Update-real-time* differs from write-real-time opacity by allowing all read-only transactions to break real-time order. *Program order opacity* requires real-time order only for transactions executed by the same process. In this sense, it is similar to virtual time opacity [13]. However, unlike virtual time opacity, program order opacity does not require that each live transaction recognizes its causal past across all processes. Finally, *arbitrary order opacity* makes no assumptions on the relative ordering of transactions. In this respect, it is similar to serializability. However, it also ensures that live transactions always observe a consistent view of the system's state.

3. Opacity

We follow the formal framework of opacity from [10], but we extend it to accommodate several new ordering relations. We also borrow some definitions from [11].

We consider a system consisting of a finite set $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ of n processes. Processes are independent and execute steps in parallel (or alternately). The system manages a set $\mathcal{X} = \{x_1, x_2, ...\}$ of transactional objects called *t-objects*. Each t-object has a unique identity and a type. Each type is defined by a *sequential specification* that consists of:

- a set Q of possible states for an object,
- an initial state $q_0 \in Q$,
- a set INV of operations that can be applied to an object,
- a set RES of possible responses an object can return, and

[3] Interestingly, the majority of correctness criteria discussed below were formulated with a local environment in mind, where communication between processes is relatively inexpensive. Therefore they are not suitable for using in a distributed environment.

- a transition relation $\delta \subseteq Q \times INV \times RES \times Q$.

This specification describes how the object behaves if it is accessed by one operation at a time. If $(q, op, res, q') \in \delta$, it means that a possible result of applying operation op to an object in state q is that the object moves to state q' and returns the response res to the process that invoked op. For simplicity, we assume that operation arguments are encoded in the operation itself.

We say that an operation op is *updating* for a given state $q \in Q$, if and only if there exists $(q, op, res, q') \in \delta$, such that $q \neq q'$. We say that op is *read-only*, if and only if there does not exist a state q for which op is updating.

We distinguish a set $\mathcal{T} = \{T_1, T_2, ...\}$ of transactions. A transaction is an abstract notion fully controlled by some process. For convenience, we say that a transaction T_k performs some action when a given process executes this action as part of the transaction T_k. T-objects can only be accessed through the TM interface (see below) and by any transaction T_k. A transaction that only executes read-only operations is called a *read-only transaction*. Otherwise, we say that it is an *updating transaction*. In general, it is impossible to tell whether a transaction is read-only before it finishes its execution. However, we distinguish a special class of transactions called *declared read-only* (*DRO*), which are known *a priori* to be read-only (they are allowed to execute only read-only operations on t-objects). Then, for any such transaction T_k, we write $DRO(T_k) = true$. Every transaction T_k for which $DRO(T_k) = true$ is read-only, but the opposite is not necessarily true.

We consider a TM interface consisting of the following operations:

- $texec(T_k, x.op) \rightarrow \{v, A_k\}$ which executes an operation op on a t-object x, of some type $T = (Q, q_0, INV, RES, \delta)$, within a transaction T_k and as a result produces a return value $v \in RES$ or the special value A_k;
- $tryC(T_k) \rightarrow \{A_k, C_k\}$ which attempts to commit a transaction T_k, and returns the special values A_k or C_k;
- $tryA(T_k) \rightarrow A_k$ which aborts a transaction T_k and always returns A_k.

The special value A_k, that can be returned by all the operations, indicates that the transaction T_k has been aborted. The value C_k returned by the operation $tryC(T_k)$ means that T_k had indeed committed. For any t-object of type $T = (Q, q_0, INV, RES, \delta)$, $A_k \notin RES$, and $C_k \notin RES$. A response event with a return value A_k or C_k is called, respectively, an *abort event* or *commit event* (of transaction T_k). The commit or abort events of a transaction T_k are always the last events for T_k.

When a process p_i executes a TM operation op, it invokes an event $inv_i(op)$ and expects a response event $resp_i(v)$. A pair of such events is called a *(completed) operation execution* and is denoted by $op \rightarrow_i v$. An invocation event that is not followed by a response event is called a *pending operation execution*.

We model the system execution as a (totally ordered) sequence of events called a *history*. Naturally, histories respect program order (events executed by the same process are ordered according to their execution order), and also causality between events across processes (if two events executed in the system are causally related, one will precede the other in the history). Events that happen in parallel (in separate processes), and that are not causally dependent, can appear in a history in an arbitrary order.[4] For any history H, we denote by $H|p_i$ the restriction of H to events issued or received by the process p_i. Similarly, we denote by $H|T_k$ the restriction of H to events concerning T_k, i.e. invocation events of the operations $texec(T_k, x.op)$, $tryC(T_k)$, $tryA(T_k)$ or their corresponding response events (for any t-object x and operation op). We say that a transaction T_k is in H if $H|T_k$ is not empty. Let x be any t-object. We denote by $H|x$ the restriction of H to events concerning x, i.e. the invocation events of any operation $texec(T_k, x.op)$ and their corresponding response events (for any transaction T_k and operation op on x).

A history H is said to be *well-formed* if, for every process p_i, $H|p_i$ is a (finite or infinite) sequence of operation executions, possibly ending with a pending operation execution. We consider only well-formed histories.

Let H be any history. We say that a transaction T_k is *committed* in H, if $H|T_k$ contains operation execution $tryC(T_k) \rightarrow_i C_k$ (for some process p_i). We say that transaction T_k is *aborted* in H, if $H|T_k$ contains response event $resp_i(A_k)$ from any TM operation (for some process p_i). If an aborted transaction T_k contains an invocation of the operation $tryA(T_k)$ it is said to be *aborted on demand*, otherwise we say that transaction T_k is *forcibly aborted* in H. A transaction T_k in H that is committed or aborted is called *completed*. A transaction that is not completed is called *live*. A transaction T_k is said to be *commit-pending* in a history H, if $H|T_k$ has a pending operation $tryC(T_k)$ (T_k invoked the operation $tryC(T_k)$, but has not received any response from this operation).

Let H be any history. We say that H is *completed* if every transaction T_k in H is completed. A *completion* of a history H is any (well-formed) complete history H' such that:

1. H is a prefix of H', and
2. for every transaction T_k in H, sub-history $H'|T_k$ is equal to one of the following histories:
   - $H|T_k$, when T_k is completed, or
   - $H|T_k \cdot \langle tryA(T_k) \rightarrow_i A_k \rangle$, for some process p_i, when T_k is live and there is no pending operation in $H|T_k$, or
   - $H|T_k \cdot \langle resp_i(A_k) \rangle$, when T_k is live and there is a pending operation in $H|T_k$ invoked by some process p_i, or
   - $H|T_k \cdot \langle resp_i(C_k) \rangle$, when T_k is commit-pending for some process p_i.

Let T_i and T_j be any two transactions in some history H, where T_i is completed. We define the following order relations on transactions in H:

- *real-time order* $\prec_H^r$ — we say that $T_i \prec_H^r T_j$ (read as T_i *precedes* T_j) if the last event of T_i precedes the first event of T_j. We call $\prec_H^r$ the *real-time order* relation in H;
- *commit-real-time order* $\prec_H^c$ — we say that $T_i \prec_H^c T_j$ if (1) $T_i \prec_H^r T_j$, and (2) both T_i and T_j are committed, or both T_i and T_j are executed by the same process p_i. We call $\prec_H^c$ the *commit-real-time order* relation in H;
- *write-real-time order* $\prec_H^w$ — we say that $T_i \prec_H^w T_j$ if (1) $T_i \prec_H^r T_j$, and (2) both T_i and T_j are not declared read-only and are committed, or both T_i and T_j are executed by the same process p_i. We call $\prec_H^w$ the *write-real-time order* relation in H;
- *update-real-time order* $\prec_H^u$ — we say that $T_i \prec_H^u T_j$ if (1) $T_i \prec_H^r T_j$, and (2) both T_i and T_j are updating and are

[4] An alternative approach to model the system execution would be to employ *partially ordered sets* (or simply *posets*). In fact, this approach is equivalent to ours, because a poset can be represented by a set of totally ordered histories (and we always analyze all the possible histories a given system can produce). We argue, that an approach based on totally ordered histories is more elegant, because it better matches the sequential specifications used for t-objects. We also want to stay close to, and remain compatible with, the original formal framework of opacity presented in [10].

committed, or both T_i and T_j are executed by the same process p_i. We call $\prec^u_H$ the *update-real-time order* relation in H;

- *program order* $\prec^p_H$ — we say that $T_i \prec^p_H T_j$ if $T_i \prec^r_H T_j$ and both T_i and T_j are issued by the same process p_i. We call $\prec^p_H$ the *program order* relation in H.
- *arbitrary order* $\prec^a_H$ — equivalent to $\emptyset$. Never $T_i \prec^a_H T_j$ holds true. We call $\prec^a_H$ the *arbitrary order* relation in H.

Let H, H' be two histories. We say that H' respects the $\diamond$ order of H iff $\prec^\diamond_H \subseteq \prec^\diamond_{H'}$. For any history H the following holds: $\emptyset = \prec^a_H \subseteq \prec^p_H \subseteq \prec^u_H \subseteq \prec^w_H \subseteq \prec^c_H \subseteq \prec^r_H$.

We say that T_i and T_j are *concurrent* if neither $T_i \prec^r_H T_j$ nor $T_j \prec^r_H T_i$. We say that any history H is *t-sequential* if H has no concurrent transactions.

Let S be any completed t-sequential history, such that every transaction in S, possibly except the last one, is committed. We say that S is *t-legal* if, for every t-object x, the subhistory $S|x = \langle texec(T_k, x.op_1) \rightarrow_i res_1, texec(T_l, x.op_2) \rightarrow_j res_2, ...\rangle$ (for any processes p_i, p_j, ..., and for any transactions T_k, T_l, ...) satisfies the sequential specification of x, $(Q, q_0, INV, RES, \delta)$, in the following sense: there exists a sequence of states $q_1, q_2, ...$ in Q, such that $(q_{i-1}, op_i, res_i, q_i) \in \delta$ for any i.

Let S be any completed t-sequential history. We denote by $visible_S(T_k)$ the longest subsequence S' of S such that, for every transaction T_i in S', either (1) $i = k$, or (2) T_i is committed and T_i precedes T_k. We say that a transaction T_k in S is *t-legal* in S, if the history $visible_S(T_k)$ is t-legal.

We say that histories H and H' are *equivalent*, and we write $H \equiv H'$, if for every transaction T_k in $\mathcal{T}$, $H|T_k = H'|T_k$.

Definition 1. *A finite history H is* final-state $\diamond$ opaque *if there exists a t-sequential history S equivalent to any completion of H, such that:*

1. *every transaction T_k in S is t-legal in S, and*
2. *S respects the $\diamond$ (order) of H.*

Definition 2. *A history H is* $\diamond$ opaque *if every finite prefix of H is final-state $\diamond$ opaque.*

In the above two definitions $\diamond$ can be either real-time, commit-real-time, write-real-time, update-real-time, program order, or arbitrary order. Therefore we obtain a whole family of $\diamond$ opacity properties. Real-time opacity is equivalent to opacity. By substituting real-time with weaker ordering guarantees we obtain gradually weaker properties with arbitrary order opacity being the weakest one.

Real-time opacity, which is equivalent to the original definition of opacity [10], requires that all transactions, regardless of their state of execution (live, aborted, commit-pending or committed) always observe a consistent and the most recent view of the system. Commit-real-time opacity relaxes opacity, by restricting the real-time order to only committed transactions (thus allowing aborted transactions to observe stale, but consistent data). Write-real-time opacity and update-real-time opacity additionally relax the real-time order requirement on transactions that, respectively, are known *a priori* to be read-only (are declared read-only), or do not perform any updating operations (are read-only). Program order opacity ensures that transactions respect program order (i.e. the order of execution of all local transactions has to be respected across all processes). Finally, arbitrary order opacity imposes no requirements on the order of transactions' execution, as long as all transactions are t-legal.

Write-real time opacity is suitable only for systems that can distinguish between transactions that did not perform any updating operations and transactions known *a priori* to be read only (only for the latter ones the DRO predicate holds). In such systems, the additional information about transactions can be either provided by the programmer or can be deduced prior to a transaction execution from the transaction code itself. By manually marking some transactions as declared read-only, a programmer can decide whether a read-only transaction T_k may read-stale data ($DRO(T_k)$ holds) or has to respect real-time order ($DRO(T_k)$ does not hold). We can make the following two observations. Firstly, given a history H which is write real-time opaque, if there are no transactions for which the DRO predicate holds, H is also commit-real-time opaque. Secondly, given a history H that is update real-time opaque, if for all read-only transactions the predicate DRO holds, H is also write real-time opaque.

Figure 1 illustrates the relations between the members of the $\diamond$ opacity family by example. It depicts four histories (two variants of history H_b can be deduced depending on the value of $DRO(T_3)$). Each history represents a case when one property is satisfied while another, a stronger one, is not.

Histories H_a and H_b represent our main motivation: enabling aborted and read-only transactions to read from a stale (but consistent) snapshot. Let us first consider H_a. Transactions T_1 and T_2 access the same t-object x. T_1 precedes T_2, however T_2 reads a stale value of x, and subsequently aborts. The only possible serialization of H_a in which all transactions are t-legal is $\langle H_a|T_2 \cdot H_a|T1\rangle$. This serialization does not respect the real-time order, as clearly $T_1 \prec^r_{H_a} T_2$. Therefore, H_a breaks (real-time) opacity. It satisfies, however, commit-real-time opacity, because T_2 is aborted and it may observe stale data.

In history H_b, transaction T_3, which does not perform any updating operations, is preceded by transaction T_2. However, T_3 does not observe the operation $x.wr_2(2)$ of T_2, as its operation $x.rd_3$ returns the value written by T_1. Therefore, H_b also breaks real-time opacity. Moreover, it breaks commit-real-time opacity. On the other hand, H_b satisfies write-real-time opacity when $DRO(T_3)$ holds, and update-real-time opacity when $DRO(T_3)$ does not hold.

In history H_c, similarly as in the previous example, T_3 does not obey the real-time order. This time, however, T_3 is an updating transaction. This causes the history to satisfy only program order opacity and not update-real-time opacity, nor any stronger property. Finally, in history H_d, even the program order is not preserved, as p_2's transaction T_3 does not observe the effects of another transaction (T_2) executed by p_2 earlier. This history, however, satisfies arbitrary order opacity, as the transactions T_2 and T_3 can be reordered, yielding an equivalent legal execution. This trait makes arbitrary order opacity similar to serializability.

4. Deferred Update Replication

In this section, we briefly describe a basic version of the Deferred Update Replication protocol and then show why it guarantees update-real-time opacity. We follow the description of DUR from [16].

4.1 Specification

DUR typically assumes full replication of shared data items (or shared objects), on which transactions operate. In our pseudocode, which is presented in Algorithm 1, each shared object is identified by a unique value of a special type *objectId*. For simplicity, we assume that each shared object can only be read or written to.

Transactions are submitted to the system by clients. Each request consists of three elements: $code$, which specifies the operations to be executed within a transaction, $args$, which holds the arguments needed for the code execution and $clock$, a special integer value necessary for ensuring that all earlier requests issued by the client are serialized before the most recent client's request.

Each process maintains two global variables. The first one, LC, represents the logical clock which is incremented every time a process applies updates of a new transaction (line 57). LC allows

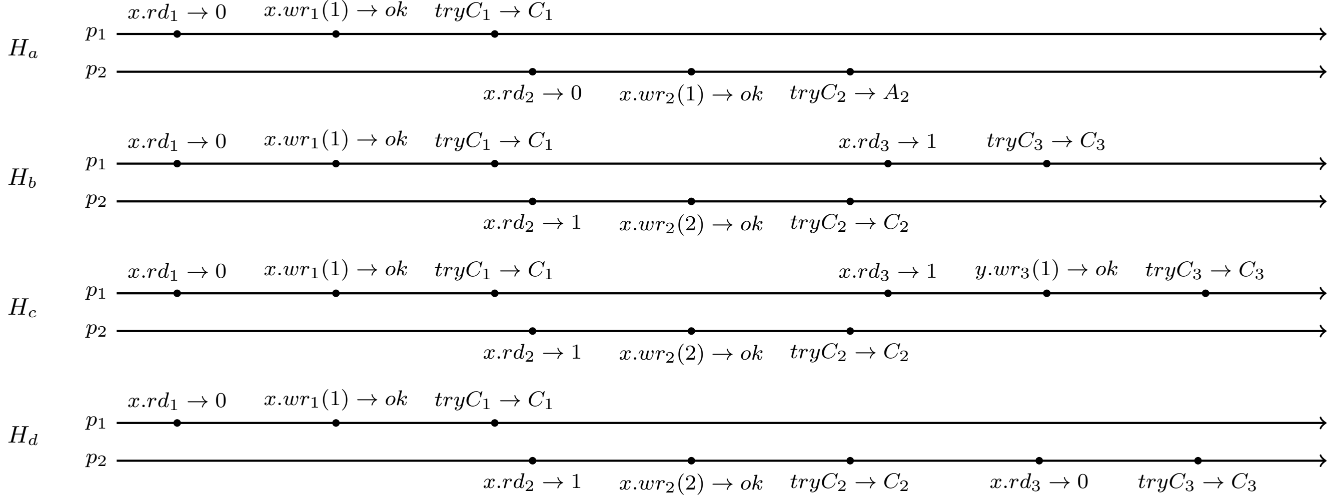

Figure 1. Example histories for two processes p_1 and p_2. History H_a is commit-real-time opaque, but not real-time opaque. History H_b is update-real-time opaque, but not commit-real-time opaque. Additionally, H_b is write-real-time opaque, iff $DRO(T_3)$ in H_b. History H_c is program order opaque, but not update-real-time opaque. History H_d is arbitrary order opaque, but not program order opaque.

the process to track whether its state is recent enough to execute the client's request (line 19). Additionally, LC is used to mark the start and the end of the transaction execution (lines 24 and 54). The transaction's start and end timestamps, stored in the *transaction descriptor* (line 16), allow us to reason about the precedence order between transactions. Let H be some execution of the algorithm, T_i and T_j some transactions in H and t_i and t_j be their transaction descriptors, respectively. Then, $t_i.end \leq t_j.start$ holds only if $T_i \prec_H^r T_j$.[5] The second variable, *Log*, is a set used to store the transaction descriptors of committed transactions. Maintaining this set is necessary to perform transaction certification.

The DUR algorithm detects any conflicts among transactions by checking whether a given transaction T_k that is being certified read any stale data. The latter occurs when T_k read any shared objects that have been modified by a concurrent but already committed transaction. For this purpose, DUR traces the accesses to shared objects independently for each transaction. The identifiers of objects that were read and the modified objects themselves are stored in private, per transaction, memory spaces: *readset* and *updates*. On every read, an object's identifier is added to the *readset* (line 30). Similarly, on every write a pair of the object's identifier and the corresponding object is recorded in the *updates* set (line 28). Then, the CERTIFY function compares the given *readset* against the *updates* of all the committed transactions in *Log* that are concurrent with the tested transaction. If it finds any non-empty intersection of the sets, the outcome is negative. Otherwise, it is positive (no conflicts detected, the transaction is certified successfully). Note that every time a transaction reads some shared object, a check against conflicts is performed (line 31). This way T_k is guaranteed to always read from a consistent snapshot. When a conflict is detected, T_k is aborted and forced to retry.

When a transaction's code completes, the COMMIT operation (line 35) is used to end the transaction and initiate the committing phase, which can be explained as follows. If T_k is a read-only transaction (T_k did not modify any objects), it can commit straight away, without performing any further conflict checks or process synchronization (lines 36–37). A read-only transaction does not need to perform certification as the possible conflicts would have been detected earlier, upon read operations (line 31). On the other hand, for updating transactions, first, the local certification takes place (line 38), which is not mandatory, but allows the process to detect conflicts earlier, and thus sometimes avoid costly network communication. Next, the transaction's descriptor containing *readset* and *updates* is broadcast to all processes using TO-BROADCAST (line 40). The message is delivered in the main thread, where the final certification takes place (line 52). Upon successful certification of transaction T_k, processes apply the updates performed by T_k and commit it (lines 54–57). Otherwise, T_k is rolled back and reexecuted by the same process.

To manage the control flow of a transaction, the programmer can use two additional operations: ROLLBACK and RETRY, whose semantics is similar as in transactional memory systems. The ROLLBACK operation (line 46) stops the execution of a transaction and revokes all the changes it performed so far. The RETRY operation (line 48) forces a transaction to rollback and restart.

For clarity, we made several simplifications. Firstly, note that the operations on LC (lines 24, 54, 57), *Log* (lines 10 and 55) and the accesses to transactional objects (lines 7 and 56) have to be synchronized. For simplicity, a single global lock is used. For better performance, the implementation can rely on fine-grained locks. Secondly, in our pseudocode, *Log* can grow indefinitely. In reality, *Log* can easily be kept small by garbage collecting information about the already committed transactions that ended before the oldest live transaction started its execution in the system.

In the presented algorithm, we use the same certification procedure for both the certification test performed upon every read operation (line 31) and the certification test that happens after a transaction descriptor is delivered to the main thread (line 52). In practice, however, doing so would be very inefficient. It is because for every read operation, we check for the conflicts against all the concurrent transactions (line 10), thus performing much of the same work again and again. However, these repeated actions can be easily avoided by associating the accessed shared objects with version

[5] Moreover, if both T_i and T_j are committed updating transactions, $T_i \prec_H^r T_j$ and $t_i.end > t_j.start$, then T_i and T_j must not be in conflict (as otherwise T_j would be aborted).

Algorithm 1 Deferred Update Replication for process p_i

```
1: integer LC ← 0
2: set Log ← ∅
3: function GETOBJECT(txDescriptor t, objectId oid)
4:     if (oid, obj) ∈ t.updates then
5:         value ← obj
6:     else
7:         value ← retrieve object oid
8:     return value
9: function CERTIFY(integer start, set readset)
10:    lock { L ← {t ∈ Log : t.end > start} }
11:    for all t ∈ L do
12:        writeset ← {oid : ∃(oid, obj) ∈ t.updates}
13:        if readset ∩ writeset ≠ ∅ then
14:            return failure
15:    return success
Thread q on request r from client c (executed on one replica)
16: txDescriptor t ← ⊥    // type: record (id, start, end, readset, updates)
17: response res ← ⊥
18: upon INIT
19:     wait until LC ≥ r.clock
20:     raise TRANSACTION
21:     return (id, LC, res) to client c
22: upon TRANSACTION
23:     t ← (a new unique id, 0, 0, ∅, ∅)
24:     lock { t.start ← LC }
25:     res ← execute r.code with r.args
26:     COMMIT()
27: upon WRITE(objectId oid, object obj)
28:     t.updates ← {(oid′, obj′) ∈ t.updates : oid′ ≠ oid}
                    ∪{(oid, obj)}
29: upon READ(objectId oid)
30:     t.readset ← t.readset ∪ {oid}
31:     lock { if CERTIFY(t.start, {oid}) = failure then
32:                raise RETRY
33:            else
34:                return GETOBJECT(t, oid) }
35: procedure COMMIT
36:     if t.updates = ∅ then
37:         return to INIT
38:     if CERTIFY(t.start, t.readset) = failure then
39:         raise RETRY
40:     TO-BROADCAST t
41:     wait for outcome
42:     if outcome = failure then
43:         raise RETRY
44:     else                                    // outcome = success
45:         return to INIT
46: upon ROLLBACK
47:     stop executing r.code and return to INIT
48: upon RETRY
49:     stop executing r.code
50:     raise TRANSACTION
The main thread of DUR (executed on all replicas)
51: upon TO-DELIVER (txDescriptor t)
52:     outcome ← CERTIFY(t.start, t.readset)
53:     if outcome = success then
54:         lock { t.end ← LC
55:                Log ← Log ∪ {t}
56:                apply t.updates
57:                LC ← LC + 1 }
58:     if transaction with t.id executed locally by thread q then
59:         pass outcome to thread q
```

number equal to the value of LC at the time the objects were most recently modified.

4.2 Correctness

Now we show that DUR indeed satisfies update-real-time opacity.

Theorem 1. *Deferred Update Replication satisfies update-real-time opacity.*

Proof sketch. In order to prove that DUR satisfies update-real-time opacity, we have to show that for every history H produced by DUR, there exists a sequential history S equivalent to some completion of H, such that S respects the update-real-time order of H and every transaction T_k in S is t-legal in S.

Informally, we have to prove that for any execution history H of DUR there exists a sequential history S such that: (1) all updating transactions in S are ordered in a way that respects the real-time order of their original execution, (2) S reflects the transaction execution order of every process (program order) and (3) every transaction T_k (no matter its state of execution) always observes a consistent state of the system, i.e., in S, every read operation of T_k on each t-object x returns the value stored by the most recent preceding write operation on x of some committed transaction (or T_k itself).

Now, we show how to construct a sequential history S from any execution H, such that S satisfies (1), (2) and (3). Let $update : \mathbb{N} \rightarrow \mathcal{T}$ be a function that maps the values LC have taken during the execution to committed updating transaction which set that particular value. Let $S = \langle H|update(1) \cdot H|update(2) \cdot ...\rangle$. This way S includes the operations of all the committed updating transactions in H. Now, let us add the rest of transactions from H to S in the following way. For every such a transaction T_k with a transaction descriptor t_k, find a committed updating transaction T_l with a transaction descriptor t_l in S, such that $t_k.start = t_l.end$, and insert $H|T_k$ immediately after T_l's operations in S. If there is no such transaction T_l ($t_k.start = 0$), then add $H|T_k$ to the beginning of S. In case of a process that executed multiple such transactions with the same $start$ timestamp, rearrange them in S according to the order in which they were executed by the process.

Let us now see why S satisfies (1). All committed updating transactions are serialized in S according to the order in which they modified LC upon commit. This order is established by TOB. Consider two committed updating transactions T_i and T_j in S, such that $T_i \prec^r_H T_j$. It means that, the first operation of T_j in H must have appeared after the commit of T_i. Therefore, T_i must have been broadcast (and delivered by the majority of processes) by TOB before T_j was broadcast.

Now, let us consider (2). Trivially, S respects program order for all committed updating transactions ($\prec^p_H \subseteq \prec^r_H$). In order to show why S respects program order also for other transactions, let us consider two transactions T_i and T_j (with transaction descriptors t_i and t_j) executed by the same process p_i. Since, a process can execute only one transaction at a time, either $T_i \prec^r_H T_j$ or $T_j \prec^r_H T_i$. Without loss of generality, let us assume the former holds. Since, LC increases monotonically during the execution, $t_i.start \leq t_j.start$. If $t_i.start < t_j.start$, then, according to the procedure described earlier, T_i was inserted after a committed updating transaction T_k with an end timestamp equal to $t_i.start$ (or T_i was inserted at the beginning of S), and T_j was inserted after a committed updating transaction T_l with an end timestamp equal to $t_j.start$. Thus, T_k must precede T_l in S (or T_k does not

exist), and so T_i has to precede T_j as well. On the other hand, if $t_i.start = t_j.start$, then, according to the procedure, we rearrange T_i and T_j in S according to the order in which they were executed by p_i, thus explicitly satisfying (2).

The proof of (3) is based on the following two observations. Firstly, every process maintains only the most recent value of every t-object. It means that if some transaction T_k executed by some process p_i reads a t-object x, the read value either is the initial state of x, has been written by T_k, or has been written by the most recently committed transaction that modified x and whose updates have been applied to the state of p_i. Secondly, before any transaction T_k can read some t-object, it has to ensure that it is not about to read from a concurrent but already committed transaction. An attempt to read an inconsistent value results in a rollback and restart T_k. Since the order of updating transactions in S (and the order of every write operation) is equivalent to the order in which updates are applied to the local state, (3) is satisfied for every transaction in H.

Since for every execution history H of DUR we can find an equivalent sequential history S that satisfies (1), (2) and (3), DUR guarantees update-real-time opacity. □

5. Conclusions

In this paper we tackled the problem of opacity being an inadequate correctness criterion for Deferred Update Replication protocol, typically used for achieving consistency in transactional distributed systems. Our new property, called update-real-time opacity, precisely describes the characteristics of DUR by relaxing real-time order requirements for transactions that do not perform any updating operations. Update-real-time opacity is suitable not only for describing the behaviour of DUR but also of similar replication protocols such as Postgres-R and EDUR.

We described update-real-time opacity as a member of a new ◇ opacity family of properties whose members relax time ordering requirements of opacity to various extent. This way we are able to formalize the behaviour of a wider class of strongly consistent transactional systems.

In the future, we plan to extend the family of properties based on opacity to account for other replication protocols, including the eventually consistent ones.

Acknowledgments

The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.

References

[1] P. A., Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[2] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Proc. of Euro-Par '97*, Aug. 1997.

[3] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *Proc. of ICDCS '13*, 2013.

[4] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9), 1991.

[5] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication - Theory and Practice*, volume 5959 of *LNCS*. Springer, 2010.

[6] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proc. of PRDC '09*, Nov. 2009.

[7] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36 (4), Dec. 2004.

[8] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5), 2013.

[9] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of PPoPP '08*, Feb. 2008.

[10] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[11] R. Guerraoui and E. Ruppert. Linearizability is not always a safety property. In *Proc. of NETYS '14*. May 2014.

[12] R. Hansdah and L. Patnaik. Update serializability in locking. In *Proc. of ICDT '86*. Sept. 1986.

[13] D. Imbs, J. R. G. De Mendivil Moreno, and M. Raynal. On the Consistency Conditions of Transactional Memories. Research Report PI 1917, 2008. URL `https://hal.inria.fr/inria-00350131`.

[14] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. VLDB '00*, Sept. 2000.

[15] T. Kobus, M. Kokociński, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proc. ICDCS '13*, July 2013. .

[16] T. Kobus, M. Kokociński, and P. T. Wojciechowski. Introduction to transactional replication. In R. Guerraoui and P. Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *LNCS*. Springer, 2015.

[17] M. Kokociński, T. Kobus, and P. T. Wojciechowski. Make the leader work: Executive deferred update replication. In *Proc. of SRDS '14*, Oct. 2014.

[18] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[19] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of Euro-Par '98*, Sept. 1998.

[20] F. Pedone, R. Guerraoui, and André. The database state machine approach. *Distributed and Parallel Databases*, 14(1), July 2003.

[21] D. Sciascia and F. Pedone. RAM-DUR: In-Memory Deferred Update Replication. In *Proc. of SRDS '12*, Oct. 2012.

[22] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Proc. of DSN '12*, June 2012.

[23] 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. .