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
Replcation 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. 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 transactions 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 anyway, 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. 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, that 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.
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 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.\footnote{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.}

The following properties maintain a global serialization only for some of the transactions. Extended update serializability [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 serializability 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 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 \( P = \{p_1, p_2, \ldots, p_n\} \) of \( n \) processes. Processes are independent and execute steps in parallel (or alternately). The system manages a set \( X = \{x_1, x_2, \ldots\} \) 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
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 $T = \{T_1, T_2, \ldots\}$ 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:

- $txexec(T_k, x, op) \rightarrow \{v, A_k\}$ 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 $resp_{res}(v)$. 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. 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 $txexec(T_k, x, op)$, $tryC(T_k)$, $tryA(T_k)$ 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 $txexec(T_k, x, op)$ 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 $tryC(T_k) \rightarrow C_k$ (for some process $p_i$). We say that transaction $T_k$ is aborted in $H$, if $H[T_k]$ contains response event $resp_{res}(A_k)$ 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 committed. 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
   - $H[T_k] \cdot (tryA(T_k) \rightarrow A_k)$, for some process $p_i$, when $T_k$ is live and there is no pending operation in $H[T_k]$, or
   - $H[T_k] \cdot (resp_{res}(A_k))$, when $T_k$ is live and there is a pending operation in $H[T_k]$ invoked by some process $p_i$, or
   - $H[T_k] \cdot (resp_{res}(C_k))$, 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 $\prec_H^{cr}$ — we say that $T_i \prec_H^{cr} T_j$ 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 $\prec_H^{cr}$ the commit-real-time order relation in $H$;
- write-real-time order $\prec_H^{wr}$ — we say that $T_i \prec_H^{wr} T_j$ 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 $\prec_H^{wr}$ the write-real-time order relation in $H$;
- update-real-time order $\prec_H^{ur}$ — we say that $T_i \prec_H^{ur} T_j$ if (1) $T_i \prec_H^r T_j$, and (2) both $T_i$ and $T_j$ are updating and are
committed, or both \(T_i\) and \(T_j\) are executed by the same process \(p_i\). We call \(\prec_p\) the update-real-time order relation in \(H\):

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

Let \(H, H'\) be two histories. We say that \(H'\) respects the \(\diamond\) order of \(H\) iff \(\prec H H' \subseteq \prec H H\). For any history \(H\) the following holds: \(\emptyset = \prec H H' \subseteq \prec H H\) if and only 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 \(S[x = (\texttt{exec}(T_i, x, op_1) \rightarrow res_1, \texttt{exec}(T_i, x, op_2) \rightarrow res_2, \ldots)\) for any transactions \(p_i, p_j, \ldots\) and for any transactions \(T_k, T_l, \ldots\) satisfies the sequential specification of \(x\), \((Q, q_0, INV, RES, \delta)\), in the following sense: there exists a sequence of states \(q_1, q_2, \ldots\) in \(Q\), such that \((q_{i-1}, op_i, res_i, q_i) \in \delta\) for every \(i\).

Let \(S\) be any completed t-sequential history. We denote by \(\text{visible}_S(T_k)\) the longest subsequence \(S'\) of \(S\) such that, for every transaction \(T_k\) 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 \(\text{visible}_S(T_k)\) is t-legal.

We say that histories \(H, H'\) are equivalent, and we write \(H \equiv H'\), if for every transaction \(T_k\) in \(H\), \(H[T_k] = H'[T_k]\).

**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 \(\diamond\) (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 order 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 \(T_i\) may read-stale data (\(\text{DRO}(T_i)\) holds) or has to respect real-time order (\(\text{DRO}(T_i)\) 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 \(H_0\) can be deduced depending on the value of \(\text{DRO}(T_3)\)). Each history represents a case when one property is satisfied while another, a stronger one, is not.

Histories \(H_0\) and \(H_0'\) represent our main motivation: enabling aborted and read-only transactions to read from a stale (but consistent) snapshot. Let us first consider \(H_0\). Transactions \(T_1\) and \(T_2\) access the same t-object \(x\). \(T_2\) precedes \(T_1\) however \(T_2\) reads a stale value of \(x\), and subsequently aborts. The only possible serialization of \(H_0\) in which all transactions are t-legal is \((H_0[T_2], H_0[T_1])\). This serialization does not respect the real-time order, as clearly \(T_1 \sim H_0 T_2\). Therefore, \(H_0\) breaks (real-time) opacity. It satisfies, however, commit-real-time opacity, because \(T_2\) is aborted and it may observe stale data.

In history \(H_0\), transaction \(T_3\), which does not perform any updating operations, is preceded by transaction \(T_2\). However, \(T_2\) does not observe the operation \(x.wrz(2)\) of \(T_2\), as its operation \(x.rdz\) returns the value written by \(T_1\). Therefore, \(H_0\) also breaks real-time opacity. Moreover, it breaks commit-real-time opacity. On the other hand, \(H_0\) satisfies write-real-time opacity when \(\text{DRO}(T_3)\) holds, and update-real-time opacity when \(\text{DRO}(T_3)\) does not hold.

In history \(H_0'\), 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_0', 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
Moreover, if both $T_i$ and $T_j$ are committed updating transactions, $T_i \prec_H T_j$ 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 $p_i$

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>integer $LC \leftarrow 0$</td>
</tr>
<tr>
<td>2</td>
<td>set $Log \leftarrow \emptyset$</td>
</tr>
<tr>
<td>3</td>
<td>function $GETOBJECT(txDescriptor t, objectId oid)$</td>
</tr>
<tr>
<td>4</td>
<td>if $(oid, obj) \in t\text{.}updates$ then</td>
</tr>
<tr>
<td>5</td>
<td>$value \leftarrow obj$</td>
</tr>
<tr>
<td>6</td>
<td>else</td>
</tr>
<tr>
<td>7</td>
<td>$value \leftarrow$ retrieve object $oid$</td>
</tr>
<tr>
<td>8</td>
<td>return $value$</td>
</tr>
<tr>
<td>9</td>
<td>function $CERTIFY(integer start, set readset)$</td>
</tr>
<tr>
<td>10</td>
<td>$lock { \ L \leftarrow { t \in Log : t\text{.}end &gt; start } }$</td>
</tr>
<tr>
<td>11</td>
<td>for all $t \in L$ do</td>
</tr>
<tr>
<td>12</td>
<td>$writeset \leftarrow { oid : \exists (oid, obj) \in t\text{.}updates }$</td>
</tr>
<tr>
<td>13</td>
<td>if readset $\cap$ writeset $\neq \emptyset$ then</td>
</tr>
<tr>
<td>14</td>
<td>return failure</td>
</tr>
<tr>
<td>15</td>
<td>return success</td>
</tr>
</tbody>
</table>

Thread $q$ on request $r$ from client $c$ (executed on one replica)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>$txDescriptor t \leftarrow \bot$ if type: record (id, start, end, readset, updates)</td>
</tr>
<tr>
<td>17</td>
<td>response $res \leftarrow \bot$</td>
</tr>
<tr>
<td>18</td>
<td>upon INIT</td>
</tr>
<tr>
<td>19</td>
<td>wait until $LC \geq r\text{.}clock$</td>
</tr>
<tr>
<td>20</td>
<td>raise TRANSACTION</td>
</tr>
<tr>
<td>21</td>
<td>return (id, $LC$, $res$) to client $c$</td>
</tr>
<tr>
<td>22</td>
<td>upon TRANSACTION</td>
</tr>
<tr>
<td>23</td>
<td>$t \leftarrow$ (a new unique id, 0, 0, $\emptyset$, $\emptyset$)</td>
</tr>
<tr>
<td>24</td>
<td>$lock { t\text{.}start \leftarrow LC }$</td>
</tr>
<tr>
<td>25</td>
<td>$res \leftarrow$ execute $r\text{.}code$ with $r\text{.}args$</td>
</tr>
<tr>
<td>26</td>
<td>COMMIT()</td>
</tr>
<tr>
<td>27</td>
<td>upon WRITE(objectId oid, object obj)</td>
</tr>
<tr>
<td>28</td>
<td>$t\text{.}updates \leftarrow { (oid', obj') \in t\text{.}updates : oid' \neq oid }$</td>
</tr>
<tr>
<td></td>
<td>$\cup { (oid, obj) }$</td>
</tr>
</tbody>
</table>

29: upon READ(objectId oid)
30: $t\text{.}readset \leftarrow t\text{.}readset \cup \{ oid \}$
31: lock $(if CERTIFY(t\text{.}start, \{ oid \}) = failure then$
32: raise RETRY
33: else
34: return $GETOBJECT(t, oid)$
35: procedure COMMIT
36: if $t\text{.}updates = \emptyset$ then
37: return to INIT
38: if CERTIFY(t\text{.}start, t\text{.}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 $r\text{.}code$ and return to INIT
48: upon RETRY
49: stop executing $r\text{.}code$
50: raise TRANSACTION

The main thread of DUR (executed on all replicas)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
</table>
51: upon TO-DELIVER (txDescriptor $t$)
52: $outcome \leftarrow$ CERTIFY(t\text{.}start, t\text{.}readset)
53: if outcome = success then
54: lock $(t\text{.}end \leftarrow LC$
55: $Log \leftarrow Log \cup \{ t \}$
56: apply $t\text{.}updates$
57: $LC \leftarrow LC + 1$
58: if transaction with $t\text{.}id$ executed locally by thread $q$ then
59: pass outcome to thread $q$

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 $i$-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 $update : \mathbb{N} \rightarrow T$ be a function that maps the values $LC$ that have taken during the execution to committed updating transaction which set that particular value. Let $S = \langle H[update(1) \cdot H[update(2) \cdot \ldots)$. 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 $T_i$ with a transaction descriptor $t_i$ in $S$, such that $t_i.start = t_k.end$, and insert $H[T_k]$ immediately after $T_i$’s operations in $S$. If there is no such transaction $T_i$ ($t_i.start = 0$), 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 $T_i >> H T_j$. It means that, the first operation of $T_i$ 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 ($\prec_H < H \leq < H$). 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 $T_i < H T_j$ or $T_j < H T_i$. 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 $T_j$ in $S$ (or $T_k$ does not
updates have been applied to the state of process \( p \). We assume that \( T_i \) first
restarts its local state \( x \) and then reads \( x \). The order in which \( T_i \)
reads \( x \) is not important; the important thing is that it is not about to read from a concurrent
but already committed transaction. We attemp to read an inconsistent value results in a rollback and
restart \( T_i \). 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