Distributed Transactional Systems Cannot Be Fast

Diego Didona
EPFL
diego.didona@epfl.ch

Panagiota Fatourou
FORTH, ICS & University of Crete, CSD
faturu@csd.uoc.gr

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Jingjing Wang
EPFL
jingjing.wang@epfl.ch

Willy Zwaenepoel
EPFL & University of Sydney
willy.zwaenepoel@epfl.ch

ABSTRACT

We prove that no fully transactional system can provide fast read transactions (including read-only ones that are considered the most frequent in practice). Specifically, to achieve fast read transactions, the system has to give up support of transactions that write more than one object. We prove this impossibility result for distributed storage systems that are causally consistent, i.e., they do not require to ensure any strong form of consistency. Therefore, our result holds also for any system that ensures a consistency level stronger than causal consistency, e.g., strict serializability. The impossibility result holds even for systems that store only two objects (and support at least two servers and at least four clients). It also holds for systems that are partially replicated. Our result justifies the design choices of state-of-the-art distributed transactional systems and insists that system designers should not put more effort to design fully-functional systems that support both fast read transactions and ensure causal or any stronger form of consistency.

KEYWORDS

Causal consistency; distributed storage; fast read-only transactions; impossibility; multi-object transactions

ACM Reference Format:
Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2019. Distributed Transactional Systems Cannot Be Fast. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3323165.3323189

1 INTRODUCTION

Transactions represent a fundamental abstraction of distributed storage systems, as they facilitate the task of building correct applications. For this reason, distributed transactional storage systems are widely adopted in production environments [10, 33, 45, 48, 51, 54, 57] and actively researched in academia [19, 44, 59, 61, 64]. Because many applications exhibit read-dominated workloads [6, 17, 46, 47], read-only transactions are a particularly important building block of such systems. Hence, improving the performance of distributed read-only transactions has become a key requirement for modern such systems and a much investigated research topic [19, 40, 41]. To this end, the notion of *fast* read-only transactions has been introduced in [41] and studied in subsequent papers [23, 31, 60]. A fast read-only transaction satisfies the following three desirable properties [41]: it completes in one round of communication (*one-round*), it does not rely on blocking mechanisms (*nonblocking*), and each server communicates to the client only one value for each object that it stores locally and is being read (*one-value*). Unfortunately, despite the huge effort put on designing efficient distributed transactional systems, read-only transactions in existing systems still suffer from performance limitations. For example, systems like Spanner [19], DrTM [61], RoCoCo [44] implement read-only transactions that may require multiple rounds of communication, or rely on blocking mechanisms.

In systems that implement strong consistency, e.g., serializability [15], transactions facilitate the task of writing distributed applications by giving the illusion that concurrent operations take place sequentially. However, strong consistency comes at such a high cost [11, 36] that, over the last years, a flurry of systems has abandoned strong consistency in favor of weaker consistency models [18, 21, 39, 51]. Among them, causal consistency [2] has garnered much attention, because it was expected to hit a sweet spot in the performance versus ease-of-programming trade-off. Causal consistency has intuitive semantics and eschews the synchronization that is needed to achieve strong consistency in the presence of replicas. It is also the strongest consistency level that can tolerate network partitions without blocking operations [7, 50].

As expected, existing causally consistent storage systems achieve higher performance in comparison to strong consistency systems [3, 26, 39, 40]. However, causally consistent read-only transactions still suffer from latency overheads. In fact, state-of-the-art causally consistent storage systems either do not support fast read-only transactions [3, 26, 43, 55] (i.e., they do not exhibit all three desirable properties) or they are of restricted functionality by not providing multi-object write transactions [41].

Contributions. In this paper, we present a result proving a fundamental limitation of transactional systems. Specifically, our impossibility result states that no fully-functional, causally consistent distributed transactional system can provide fast read-only transactions (and therefore also fast read transactions). Specifically, to achieve fast read transactions, the system has to give up support

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323189

of multi-object write transactions, i.e., it can only support transactions that write at most one object. This result unveils an important trade-off between the latency attainable by read-only transactions and the functionality provided by a distributed storage system. It also shows that the inefficiency of the existing systems to achieve all the desirable properties (as described above) is not a coincidence.

Most theoretical results considered so far serializable transactions instead of causally consistent that we consider here, and this work includes a formalization of causally consistent transactional systems which is interesting in its own right. Our result holds for any system that ensures stronger consistency than causal consistency. Moreover, our result is relevant for the broad class of systems that use causal consistency as a building block to achieve their target consistency level [13] or that implement hybrid consistency models which include causal consistency [5, 35, 36, 58]. The impossibility result holds for any system that supports at least two servers and at least four clients. It holds even for systems that store only two objects (each in a different server). We prove that the impossibility result also holds for systems that are partially replicated.

To prove our impossibility result, we construct a troublesome infinite execution in which a write-only transaction that is executed solo never manages to make the values it writes visible. To do so, we inductively construct an infinite number of non-empty prefixes of the troublesome execution and prove that the written values are not yet visible after each prefix has been executed; specifically, some server has to send at least one more message before the values become visible. We argue this using indistinguishability arguments [9, 42]. The fact that, on the one hand, the constructed execution is infinite, and, on the other, that we focus on causal consistency which is a rather weak consistency condition, introduces complications that we have to cope with to get the proof.

In the case where the system has more than two servers, an extra challenge is to cope with the chains of messages through which information may be disseminated between the servers that store the written objects. This complicates the construction of the executions that we prove to be indistinguishable from the troublesome execution. To get the impossibility result for the case of a partially replicated system, an additional complication is that we have to construct an infinite sequence of server ids. These are the servers that send the necessary messages in each step for the induction to work. We also have to capture the fact that more than one servers may now respond to the same read request of a client. Due to lack of space, the general proof is provided in a technical report [22].

We study the limits of our impossibility result in all different premises. We show that if we relax any of the considered properties, then the impossibility result no longer holds.

Our impossibility result sheds light on some of the design decisions of recent systems and provides useful information to system designers. Specifically, they should not put more effort to devise a fully-functional system that supports both fast read transactions and ensures causal consistency (or any stronger consistency level).

Structure of the paper. Section 2 provides the model and useful definitions. Section 3 presents our impossibility result and discusses its limits in all different premises. Section 4 discusses related work. Section 5 provides some conclusions.

2 MODEL AND DEFINITIONS

Storage system. We consider a distributed storage system which stores a finite number of objects. There are $m > 1$ servers in the system. Each server stores a non-empty set of these objects. For simplicity, we assume that the set of objects stored in servers are disjoint. (Our result holds even if the system is *partially replicated*, i.e., if these sets are different but not disjoint, and none of them contains all the objects.)

Transactions. An arbitrarily large number of clients may read and/or write one or more objects by executing *transactions*. A *transaction* is a block of code that contains a sequence of read and write requests on objects. To prove our impossibility results, it is enough to focus on *static* transactions whose read-sets and write-sets are known from the beginning of the execution[1]. The *read-set*, R_T, of a transaction T contains the objects T reads, whereas its *write-set*, W_T, contains the objects T writes. If $W_T = \emptyset$, T is called a *read-only* transaction, whereas if $R_T = \emptyset$, T is a *write-only* transaction. We denote by $r(X)x$ a read on object X which returns *value* x and by $w(X)x$ a write of *value* x to object X. Also, we denote by $r(X)*$ a read on object X when the return value is unknown (with symbol $*$ as a place-holder). Reads and writes on objects are called *object operations*.

In typical deployments, there are many more clients than servers. Hence, allowing server-to-client out-of-band communication would result in nonnegligible overhead on the servers, which would suffer from reduced scalability and performance. For this reason, we naturally assume that to execute a transaction, a client can communicate with the servers (but not with other clients) and a server communicates with a client only to respond to a client's read or write *request*. So, *no client-to-client* communication is allowed, i.e. no client can send a message to any other client. We find evidence of the relevance of this assumption in large-scale production systems, such as Facebook's data platform [46], and in emerging systems [24, 30, 38] and architectures [37] for fast query processing, where no per-client states are maintained to avoid the corresponding overheads and to achieve the lowest latency.

System model. The system is *asynchronous*, i.e., the delay on message transmission can be arbitrarily large and the processes do not have access to a global clock. The system is modelled as an undirected graph in the standard way [9, 42]. Each node of the graph represents a process (i.e., a client or a server) whereas links connect every pair of processes. Each process is modelled as a state machine with its state containing a set of income and outcome buffers [9, Ch. 2][2]. Links do not lose, modify, inject, or duplicate messages.

Operation executions. An *implementation* of a storage system provides algorithms, for each process, to execute reads and writes in the context of transactions. A *configuration* represents an instance of the system at some point in time. In an *initial* configuration Q_{in}, all processes are in initial states and all buffers are empty (i.e., no message is in transit). There are two kinds of *events* in the system: (1) a *computation step* taken by a process, in which the process reads all messages residing in its income buffers, performs some local computation and may send (at most) one message to each

[1] It follows that our impossibility result also holds for systems of dynamic transactions.

[2] There is one income and one outcome buffer for each link incident to each process. Income and outcome buffers store the messages that are sent or received through the corresponding link, respectively.

of its neighboring processes[3], and (2) a *delivery* event, where a message is removed from the outcome buffer of the source and is placed in the income buffer of the destination. An *execution* is a sequence of events (we assume that an execution also includes the invocations and responses of transactions, as well as the invocations and responses of object operations). An execution α is *legal* starting from a configuration C, if, for every process p, every computation step taken by p in α is compatible to p's state machine (given p's state in C) and all messages sent are eventually received. Since the system is asynchronous, the order in which the events appear in an execution is assumed to be controlled by an *adversary*. A *reachable* configuration is a configuration that results from the application of a legal finite execution starting from an initial configuration. Given a reachable configuration C, we say that a computation step s by a process p *eventually occurs*, if in every legal execution starting from C (in which p takes a sufficient number of steps), p executes s. For every reachable configuration C and every legal execution α from C, we denote by $RC(C, \alpha)$ the configuration that results from the execution of α starting from C. Two executions are *indistinguishable* to a process p, if p executes the same steps in each of them. Two configurations are *indistinguishable* to p, if p is in the same state in both configurations. Given two executions α_1 and α_2, we denote by $\alpha_1 \cdot \alpha_2$ the concatenation of α_1 with α_2, i.e., $\alpha_1 \cdot \alpha_2$ is an execution consisting of all events of α_1 followed by all events of α_2 (in order).

Each client that *invokes* a transaction T may eventually return a *response*. The response consists of a value for each object in R_T, and an *ack* for each write T performs. We say that T has *completed* (or is *complete*), if the client c that invoked T has issued all read or write requests for T, and has received responses by the servers for all these requests. Note that for each read or write request that c invokes, it receives an individual response from the corresponding server. A transaction T is *active* in some configuration C if it has been invoked before C and has not yet completed. A client may have at most one active transaction at each point in time. We say that a configuration C is *quiescent* if no transaction is active at C. Let C be any reachable configuration that is either quiescent or only a single transaction T, invoked by a client c, is active at C. We say that T *executes solo* starting from C, if only c and the servers take steps after C and c does not invoke any transaction other than T after C. We say that a value x is *written* in an execution α if there exists some transaction T in α that issues $w(X)x$ for some object X. For convenience, every execution we consider, starts with the execution of two initial transactions, $T_0^{in} = (w(X_0)x_0^{in})$ (invoked by a client c_0^{in}) and $T_1^{in} = (w(X_1)x_1^{in})$ (invoked by a client c_1^{in}) that write the initial values x_0^{in} and x_1^{in} in objects X_0 and X_1, respectively.

Causal Consistency. We consider an implementation of a transactional storage system that is *causally consistent* [2, 52]. Informally, causal consistency ensures that causally-related transactions should appear, to all processes, like if they have been executed in the same order. The formal definitions below closely follow those presented for causally consistent transactional memory systems in [27].

The *history* $H(\alpha)$ of an execution α is the subsequence of α which contains only the invocations and responses of object operations (we omit α whenever it is clear from the context). A transaction T *is* in H, if H contains at least one invocation of an object operation by T. A transaction is complete in H if it is complete in α. An object operation op_1 *precedes* another object operation op_2 in α (or in $H(\alpha)$), if the response of op_1 precedes the invocation of op_2. A transaction T_1 *precedes* another transaction T_2 in α (or in $H(\alpha)$), if the last object operation invoked by T_1 precedes the first object operation invoked by T_2.

For each client c, we denote by H_c the subsequence of H containing all invocations and responses of object operations issued or received by c. Given two executions α and α', the histories $H(\alpha)$ and $H(\alpha')$ are *equivalent*, if for each client c, $H_c(\alpha) = H_c(\alpha')$. For each client c, the *program-order*, denoted by $<_{H|c}$, is a relation on transactions in H_c such that for any two transactions T_1, T_2, $T_1 <_{H|c} T_2$ if and only if T_1 *precedes* T_2 in H_c.

We denote by $complete(H)$ the subsequence of all events in H issued and received by complete transactions[4].We denote by $comm(H)$ a history that extends H: (1) $comm(H) = H \cdot H''$, i.e., H is a prefix of $comm(H)$, (2) H'' contains only responses for those write operations in H for which there is no response.

An execution σ is *sequential* if for every two transactions T_1, T_2 in σ, either T_1 precedes T_2 or vice versa. We define a *sequential history* in a similar way. Consider a sequential execution σ (legal from Q_{in}) and a transaction T in σ. We say that T is *legal* in σ, if for every invocation $r(X)x$ of a read operation on any object X that T performs, the following hold: (1) if there is an invocation of a write operation by T that precedes $r(X)x$ in σ then x is the value argument of the last such invocation, (2) otherwise, if there are no transactions preceding T in σ which invoke write for object X, then x is the initial value for X, (3) otherwise, x is the value argument of the last invocation of a write operation to X, by any transaction that precedes T in σ.

Consider any sequential history S that is equivalent to H. We define a binary relation with respect to S, called *reads-from* and denoted by $<_S^r$, on *transactions* in H such that, for any two distinct transactions T_1, T_2 in H, $T_1 <_S^r T_2$ if and only if: (1) T_2 executes a read operation op that reads some object X and returns a value x for it, and (2) T_1 is the transaction in S which executes the last write operation that writes x for X and precedes T_2[5]. Each sequential history S that is equivalent to H, induces a *reads-from* relation for H. Let $\mathbf{R}_H$ be the set of all reads-from relations that can be induced for H (by considering all equivalent to H sequential histories).

We say that a sequential history S respects some relation $<$ on the set of transactions in H if it holds that for any two transactions T_1, T_2 in S, if $T_1 < T_2$, then T_1 precedes T_2 in S.

For each $<^r$ in $\mathbf{R}_H$, we define the *causal* relation for $<^r$ on transactions in H to be the transitive closure of $\bigcup_c \left(<_{H|c}\right) \cup <^r$. We denote by $\mathbf{C}_H$ the set of all causal relations in H.

Definition 1 (Causal consistency)**.** An execution α is *causally consistent* if for history $H' = comm(H(\alpha))$, there exists a causal relation $<^c$ in $\mathbf{C}_{complete(H')}$ such that, for each client c_i, there exists a sequential execution σ_i such that:

- $H(\sigma_i)$ is equivalent to $complete(H')$,
- $H(\sigma_i)$ respects the causality order $<^c$, and

[3] Note that a process may output a message to all servers storing objects it wants to access in the same computation step.

[4] We assume that in a system that supports causal or any weaker form of consistency, all transactions commit [12, 40].

[5] Formal definitions for the general case where transactions may abort are in [27].

- every transaction executed by c_i in $H(\sigma_i)$ is legal.

An implementation is *causally consistent* if each execution α it produces is causally consistent.

Intuitively, a causally consistent distributed transactional system produces executions that respect the causality order. The necessity to talk about sets of reads-from relations and sets of causal relations comes from the fact that the values written in an execution α are not distinct. If we assume that all values written in α are distinct, some of the above definitions would be simplified. Note that doing so would be sufficient to prove the impossibility result. However, we present the general formal definition above, since we believe that it might be of interest in its own right.

Progress. To avoid trivial implementations where every read-only transaction invoked by a client always returns $\perp$ or values written by the same client, we introduce the concept of value visibility.

Definition 2 (Value visibility)**.** Consider any object X and let C be any reachable configuration which is either quiescent or just a write-only transaction (by a client c_w) writing a value x into X (and possibly performing additional writes) is active in C. Value x is *visible* in C, if and only if: in every legal execution starting from C which contains just a read-only transaction T_r (invoked by a client $c \notin \{c_w, c_0^{in}, c_1^{in}\}$) that reads X, x is returned as X's value for T_r.

We focus on storage systems that ensure minimal progress for write-only transactions. This is a weak progress property ensured even by systems in which write transactions are blocking. So, our impossibility result holds also for systems that ensure any stronger progress properties for write-only transactions (and without any restriction on progress for transactions that both read and write).

Definition 3 (Minimal Progress for write-only transactions)**.** Let T_w be any write-only transaction which writes a value x into an object X (T_w may also write other objects). If T_w executes solo, starting from any reachable quiescent configuration C, then there exists a later configuration C' such that x is visible in C'.

We denote by Q_0 a reachable configuration in which both values x_0^{in} and x_1^{in}, written by the transactions T_0^{in} and T_1^{in} discussed earlier, are visible. Definition 3 implies that Q_0 is well-defined.

We next present the definition of fast read-only transactions. Definition 4 expresses the exact same properties as in the original definition which was introduced in [41] and used in [23, 31].

Definition 4 (Fast read-only transaction)**.** We say that an implementation of a distributed storage system supports *fast read-only transactions*, if in each execution α it produces, the following hold for every read-only transaction T executed in α:

(1) **Non-blocking and One-Roundtrip Property.** The client c which invoked T sends a message to all servers storing items that it wants to read and it does so in one computational step; moreover, each server performs at most one computational step to serve the request and respond to c.
(2) **One-value messages.** Each message sent from a server to a client contains only one value that has been written by some write transaction in α into an object that is stored in the server and is read by the client[6].

[6] We remark that the message may also contain some metadata (e.g., a timestamp), as long as these metadata do not reveal any information about other transactions and additional written values to the client. Since no transaction may ever abort, servers do not respond with an *abort* indication.

3 THE IMPOSSIBILITY RESULT

In this section, we prove the impossibility result:

THEOREM 1. *No causally consistent implementation of a transactional storage system supports transactions that write to multiple objects and fast read-only transactions.*

The impossibility result holds even for systems that store just two objects X_0 and X_1. For ease of presentation, we prove it for a system with two servers p_0 and p_1. However, it can easily be extended to hold for the general case, where the system has any number of servers and is partially-replicated (see [22] for the general proof). We assume that p_0 stores X_0 and p_1 stores X_1.

3.1 Outline of the Proof

We prove Theorem 1 by the way of contradiction. Assume that there exists a causally consistent implementation Π which supports fast read-only transactions and transactions that write multiple objects. Assume also that Π guarantees minimal progress for write-only transactions. We derive a contradiction by showing that there exists a *troublesome* execution which breaks minimal progress. Specifically, we construct an infinite execution α which contains just a write-only transaction $T_w = (w(X_0)x_0, w(X_1)x_1)$, invoked by some client $c_w \notin \{c_0^{in}, c_1^{in}\}$, and we show that the values x_0 and x_1 written by T_w never become visible. Intuitively, to do so, we inductively construct an infinite number of non-empty distinct prefixes of α and prove that the written values are not yet visible after each prefix has been executed. Specifically, we use indistinguishability arguments [9, 42] to prove that after the execution of each such prefix, some server has to still send at least one message before the values become visible. (In [22], we provide a table that describes the notation used throughout the proof.)

We now provide an informal, high-level outline of the proof. We start with two simple lemmas. The first shows that a transaction which reads X_0 and X_1 cannot return a subset of the values written by T_w (i.e., it returns either the new values for both objects or the initial values for both objects). The second lemma shows that if one of the values written by T_w is not visible, then both values written by T_w are not visible. We use these lemmas to determine the values read by read-only transactions in the executions that we use. Specifically, we present two execution constructions that are useful in the proof of Theorem 1. Construction 1 describes an execution in which a read-only transaction reads the initial values for X_0 and X_1, whereas Construction 2 describes an execution in which a read-only transaction reads the new values for X_0 and X_1.

The two constructions are used to build an execution γ which allows us to derive a contradiction: In γ, a read-only transaction reads a mix of old and new values for X_0 and X_1. We use γ to prove, in the induction, that the values written by T_w are not yet visible. We also prove that to make them visible, p_0 and p_1 have to exchange more messages. Therefore, after any arbitrary but finite number of steps, T_w has not yet completed and the values written by it has not yet become visible, which contradicts eventual visibility.

3.2 Useful Constructions and Lemmas

To enforce a causal relation between the values written by c_w in T_w and x_0^{in}, x_1^{in}, the troublesome execution starts with the execution of a read-only transaction $T_r^{in} = (r(X_0)*, r(X_1)*)$ by c_w (applied from

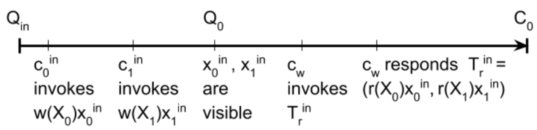


Figure 1: Configurations Q_{in}, Q_0 and C_0.

Q_0). Since T_r^{in} is a fast read-only transaction, T_r^{in} completes; since x_0^{in} and x_1^{in} are visible in Q_0 (by definition), c_w returns (x_0^{in}, x_1^{in}) for T_r^{in}. Let C_0 be the configuration in which T_r^{in} has completed and no message is in transit. The configurations Q_{in}, Q_0, C_0 are shown in Figure 1. All executions we refer to below start from C_0. We now present the two lemmas that will be useful for the constructions and the proof of Theorem 1. The first states that in an execution in which a write transaction writes new values to a set of objects, a read transaction which reads these objects, cannot see only a subset of the new values. The proof comes as an immediate consequence of the fact that Π ensures causal consistency.

Lemma 1. *Let τ be any legal execution of Π starting from C_0 which contains two transactions: client c_w invokes a write-only transaction $T_w = (w(X_0)x_0, w(X_1)x_1)$, and a client $c_r \neq c_w$ invokes a read-only transaction $T_r = (r(X_0)*, r(X_1)*)$ which completes in τ. Let v_0 and v_1 be the two values which c_r returns for T_r, i.e., $T_r = (r(X_0)v_0, r(X_1)v_1)$. Then, either $v_0 = x_0$ and $v_1 = x_1$, or $v_0 = x_0^{in}$ and $v_1 = x_1^{in}$.*

Proof of Lemma 1. To derive a contradiction, assume that there exists some execution τ of Π starting from C_0 and some $i \in \{0, 1\}$ for which c_r returns the values $v_i = x_i, v_{1-i} = x_{1-i}^{in}$ for T_r.

By causal consistency, c_r's local history in τ respects causality. So, we can totally order T_{1-i}^{in}, T_w and T_r such that (1) T_{1-i}^{in} is the last transaction which writes to X_{1-i} and precedes T_r, so T_w (which also writes X_{1-i}) precedes T_{1-i}^{in}; (2) T_w is the last transaction which writes on X_i and precedes T_r, so T_w is ordered before T_r; and (3) T_{1-i}^{in} is ordered before T_w because c_w reads the value written in T_{1-i}^{in} before it initiates T_w (as shown in Figure 1). A contradiction. □

The next lemma states that in an execution in which a write transaction T_w writes new values in a set of objects, if one of the values written by T_w is not visible at some configuration, then both values written by T_w are not visible in that configuration.

Lemma 2. *Let τ be any legal execution starting from C_0 which contains just one transaction: client c_w executes a write-only transaction $T_w = (w(X_0)x_0, w(X_1)x_1)$. Let C be any reachable configuration when τ is applied from C_0. If either x_0 or x_1 is not visible in C, then there exists at least one client $c_r \notin \{c_w, c_0^{in}, c_1^{in}\}$ such that if, starting from C, c_r executes a fast read-only transaction $T_r = (r(X_0)*, r(X_1)*)$, then c_r returns (x_0^{in}, x_1^{in}) for T_r.*

Proof of Lemma 2. To derive a contradiction, consider some configuration C, reachable when τ is applied from C_0, in which at least one of the values x_0 and x_1 is not visible, and assume that for every client $c_r \notin \{c_w, c_0^{in}, c_1^{in}\}$, if c_r invokes T_r starting from C, then c_r does not return (x_0^{in}, x_1^{in}) for T_r. Notice that since Π ensures that read-only transactions are fast, T_r completes. Then by Lemma 1, c_r returns (x_0, x_1). Since this holds for every client c_r not in $\{c_w, c_0^{in}, c_1^{in}\}$, Definition 2 implies that at C both x_0 and x_1 are visible. This contradicts the hypothesis that at least one of the two values is not visible in C. □

Notice that Lemma 2 holds for $C = C_0$, i.e., when τ is empty.

We now present Constructions 1 and 2 which are illustrated in Figure 2. Roughly speaking, the constructions illustrate two executions in which a write-only transaction T_w writes values x_0 and x_1 to objects X_0 and X_1, respectively, and a read-only transaction T_r reads X_0 and X_1. The executions are constructed so that T_r returns (x_0^{in}, x_1^{in}) in the first execution, whereas it returns (x_0, x_1) in the second. Based on these constructions, we construct, in the proof of Theorem 1, an execution where T_r returns a mix of initial and new values, allowing us to derive a contradiction. The constructions are based on a fixed $i \in \{0, 1\}$ and a client c_r that executes transaction T_r. Each time the construction is employed in the proof of Theorem 1 these parameters can be different. Although these executions are similar, we present both of them for ease of presentation.

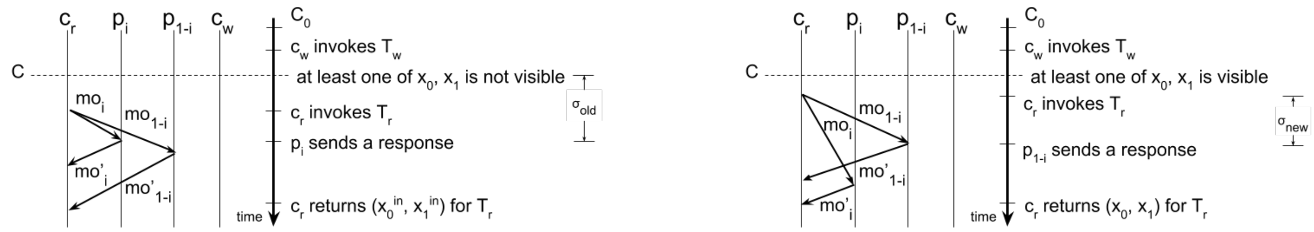
We start with an intuitive description of Construction 1 which is depicted in Figure 2(a). Assume, without loss of generality, that $i = 0$. (The construction when $i = 1$ is symmetric.) The construction produces an execution in which T_w starts executing from C_0 and runs solo up to any configuration C in which x_0 is not yet visible. Next, T_r is initiated from C and takes steps until it sends a message to both servers. The adversary schedules the receipt of these messages so that p_0 receives the message first and sends a response. Then, p_1 receives the message sent by c_r and sends back a response. Finally, c_r takes steps to collect these responses and return. We call σ_{old} the part of the execution starting from C until the point that p_0 sends a response, and γ_{old} the suffix of the execution starting from C. Lemma 2 allows us to argue that c_r returns (x_0^{in}, x_1^{in}) in γ_{old}. We next present the formalism of the construction.

Construction 1 (Construction of execution $\gamma_{old}(C, p_i, c_r)$ and execution $\sigma_{old}(C, p_i, c_r)$). Let τ be any arbitrary legal execution starting from C_0 which contains just one transaction: client c_w executes a write-only transaction $T_w = (w(X_0)x_0, w(X_1)x_1)$. Fix any $i \in \{0, 1\}$. For every configuration C that is reached when τ is applied from C_0 in which *the value x_i is not visible*, Lemma 2 implies that there exists at least one client $c_r \notin \{c_w, c_0^{in}, c_1^{in}\}$ such that if, starting from C, c_r executes a fast read-only transaction $T_r = (r(X_0)*, r(X_1)*)$, then c_r returns (x_0^{in}, x_1^{in}) for T_r. For every such client c_r, we define $\gamma_{old}(C, p_i, c_r)$ to be the execution containing all of the events described below. In $\gamma_{old}(C, p_i, c_r)$, first c_r invokes T_r starting from C. So, c_r takes steps and since it reads X_0 and X_1, c_r sends a message $mo_0(C, p_i, c_r)$ to p_0 and a message $mo_1(C, p_i, c_r)$ to p_1. Next, the adversary schedules the delivery of $mo_i(C, p_i, c_r)$ and let p_i take a step and receive $mo_i(C, p_i, c_r)$. Since T_r is a fast transaction, once p_i receives $mo_i(C, p_i, c_r)$, p_i sends a response $mo'_i(C, p_i, c_r)$ to c_r. Denote by $\sigma_{old}(C, p_i, c_r)$ the prefix of $\gamma_{old}(C, p_i, c_r)$ up until the step in which p_i sends the response. After $\sigma_{old}(C, p_i, c_r)$ has been applied from C, the adversary schedules the delivery of $mo_{1-i}(C, p_i, c_r)$, and lets p_{1-i} take steps to receive $mo_{1-i}(C, p_i, c_r)$ and send a response to c_r. Finally, c_r take steps to receive the responses from p_0 and p_1 and return a response for T_r.

By the way $\gamma_{old}(C, p_i, c_r)$ is constructed and by Lemma 2, we get the following.

Observation 1. *The following claims hold:*

(1) *Execution $\gamma_{old}(C, p_i, c_r)$ is legal from C.*



(a) Construction 1 **(b) Construction 2**

Figure 2: Illustration of Constructions 1 and 2.

(2) *Only processes p_i and c_r take steps in $\sigma_{old}(C, p_i, c_r)$, so configurations C and $RC(C, \sigma_{old}(C, p_i, c_r))$ are indistinguishable to c_w and p_{1-i}.*
(3) *The return value for T_r in $\gamma_{old}(C, p_i, c_r)$ is (x_0^{in}, x_1^{in}).*

Construction 2 is depicted in Figure 2(b). Again, assume, without loss of generality, that $i = 0$. (The construction for the case where $i = 1$ is symmetric.) The construction produces an execution in which T_w starts its execution from C_0 and runs solo up to any configuration C in which x_1 is visible. Then, T_r is initiated from C and c_r takes steps until it sends a message to each of the servers. The adversary schedules the delivery of these messages so that p_1 receives the message first and sends back a response. Then, p_0 receives the message sent by c_r and sends back a response. Finally, c_r takes steps to collect these responses and return. We call σ_{new} the part of this construction starting from C until the point that p_1 sends a response, and γ_{new} the suffix of this execution starting from C. We later argue (in Observation 2) that c_r returns (x_0, x_1) in γ_{old}. We next present the formalism of the construction.

Construction 2 (Construction of execution $\gamma_{new}(C, p_i, c_r)$ and execution $\sigma_{new}(C, p_i, c_r)$). Let τ be any legal execution starting from C_0 which contains just one transaction: client c_w executes a write-only transaction $T_w = (w(X_0)x_0, w(X_1)x_1)$. Fix any $i \in \{0, 1\}$ and let c_r be any client not in $\{c_w, c_0^{in}, c_1^{in}\}$. For every reachable configuration C when τ is applied from C_0 in which x_i *is visible*, we define $\gamma_{new}(C, p_i, c_r)$ to be the execution containing all of the events described below. In γ_{new}, c_r invokes T_r starting from C and takes steps until it sends a message $mn_0(C, p_i, c_r)$ to p_0 and a message $mn_1(C, p_i, c_r)$ to p_1. Let $C_{new}(C, p_i, c_r)$ be the resulting configuration. Next, the adversary schedules the delivery of $mn_{1-i}(C, p_i, c_r)$ and let p_{1-i} take a step to receive $mn_{1-i}(C, p_i, c_r)$. Since T_r is a fast transaction, once p_{1-i} receives $mn_{1-i}(C, p_i, c_r)$, it sends a response $mn'_{1-i}(C, p_i, c_r)$ to c_r. This sequence of steps starting from $C_{new}(C, p_i, c_r)$ to the step in which p_{1-i} sends the response is denoted by $\sigma_{new}(C, p_i, c_r)$. Next, the adversary schedules the delivery of $mn_i(C, p_i, c_r)$ and lets p_i take steps until it receives $mn_i(C, p_i, c_r)$ and sends a response to c_r. Finally, c_r takes steps to receive the responses from p_0 and p_1 and return a response for T_r.

By the way $\gamma_{new}(C, p_i, c_r)$ is constructed, by Definition 2 and by Lemma 1, we get the following.

Observation 2. *The following claims hold for $\gamma_{new}(C, p_i, c_r)$:*
(1) *Execution $\gamma_{new}(C, p_i, c_r)$ is legal from C.*
(2) *Configurations C and $RC(C, \gamma_{new}(C, p_i, c_r))$ are indistinguishable to c_w and p_i.*
(3) *The return value for T_r in $\gamma_{new}(C, p_i, c_r)$ is (x_0, x_1).*

3.3 The Infinite Execution

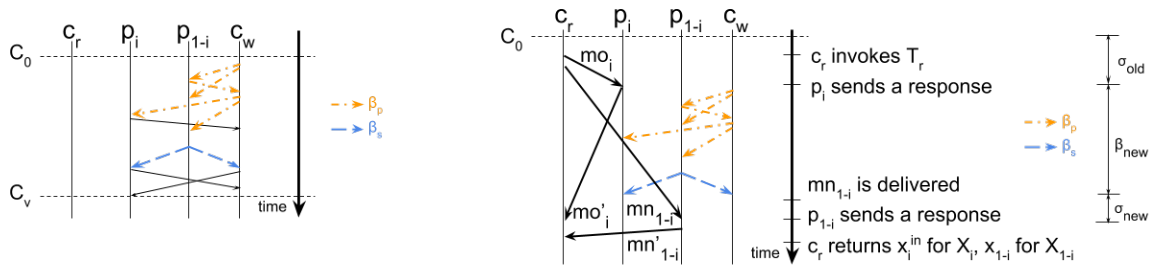
We prove Theorem 1 by constructing an infinite execution α which contains just one write-only transaction T_w and by proving that in α, the values written by T_w never becomes visible. We construct α using induction. Specifically, Lemma 3 shows that there is an infinite sequence of executions $\alpha_1, \alpha_2, \ldots$ such that, all of them are distinct prefixes of α (we let α_0 be the empty execution). Lemma 3 is comprised of two claims which hold for every integer $k \geq 0$. The first, shows that α_k contains the transmission of at least one message which is sent after the execution of α_{k-1} from C_0 (and thus, α_{k-1} is a prefix of α_k and $\alpha_{k-1} \neq \alpha_k$). The second shows that after α_k has been performed from C_0, the two values written by T_w have not yet become visible. Although the proofs of the two claims exhibit many similarities, for clarity of presentation, we have decided not to merge them into one proof.

Lemma 3. *For any integer $k \geq 1$, there exists an execution α_k, legal from C_0, in which only one transaction $T_w = (w(X_0)x_0, w(X_1)x_1)$ is executed by client c_w. Let C_k be the configuration that results when α_k is applied from C_0. Then, the following hold:*
(1) *$\alpha_k = \alpha_{k-1} \cdot \alpha'_k$, where in α'_k at least one of the following occurs:*
  - *a message is sent by $p_{k\%2}$ to $p_{(k-1)\%2}$, or*
  - *a message is sent by $p_{k\%2}$ to c_w and it holds that after c_w receives this message, c_w sends a message to $p_{(k-1)\%2}$.*

(2) *In C_k, x_0 and x_1 are not visible and T_w is still active.*

Proof of Lemma 3. By induction on k. We prove the base case together with the induction step, yet we clearly state the difference when the proof diverges. To prove the induction step, fix an integer $k > 1$ and assume that the claim holds for any j, $1 \leq j < k$.

Proof of claim 1. We start with a high-level description of the proof which is by contradiction. We come up with two executions that have the following properties. In the first execution, a read-only transaction $T_r = (r(X_0)*, r(X_1)*)$ (initiated by a client $c_r^k \notin \{c_0^{in}, c_1^{in}, c_w\}$) is executed starting from C_{k-1}. Since x_0 and x_1 are not visible neither in C_0 (since T_w has not yet started its execution in C_0), nor in C_{k-1} if $k > 1$ (by induction hypothesis), the response for T_r in this execution is (x_0^{in}, x_1^{in}). This execution is constructed based on Construction 1. In the second execution, c_r^k invokes T_r after T_w has executed solo long enough so that both values x_0 and x_1 are visible (minimal progress for write-only transactions implies that visibility of x_0 and x_1 will eventually happen). So, c_r^k returns (x_0, x_1) for T_r in this execution. The second execution is constructed based on Construction 2. We then combine parts of these two executions to get a third execution, γ. Execution γ is



(a) Execution β and its subsequences

(b) Execution γ

Figure 3: Executions β and γ. Execution β consists of the steps taken by c_w, p_i and p_{1-i} from configuration C_0, where x_0 and x_1 are not visible yet, to C_v, where these two values become visible.

constructed so that we can prove that in it, c_r^k will return the same value for $X_{k\%2}$ as c_r^k does in the first execution, and the same value for $X_{(k-1)\%2}$ as c_r^k does in the second execution. Therefore, in γ, c_r^k returns $(x_{k\%2}^{in}, x_{(k-1)\%2})$ for T_r. This contradicts Lemma 1.

We continue with the details of the proof of claim 1. To derive a contradiction, assume that the claim does not hold, i.e., we let c_w, p_0, p_1 take steps starting from C_{k-1} and assume the following:

- $p_{k\%2}$ sends no message to $p_{(k-1)\%2}$;
- $p_{k\%2}$ sends no message to c_w for which it holds that after c_w receives it, $p_{k\%2}$ sends a message to $p_{(k-1)\%2}$.

Since x_0 and x_1 are not visible neither in C_0 nor in C_{k-1}, Lemma 2 implies that there exists at least one client $c_r^k \notin \{c_w, c_0^{in}, c_1^{in}\}$ such that if, starting from C_{k-1}, c_r^k executes a read-only transaction $T_r = (r(X_0)*, r(X_1)*)$, then c_r^k returns (x_0^{in}, x_1^{in}) for T_r. We derive the contradiction by constructing the execution γ, in which, in addition to T_w, c_r^k executes such a read-only transaction T_r, and by showing that γ contradicts Lemma 1.

To construct γ, we need to define an execution β and a subsequence β_{new} of it. Specifically, β_{new} is utilized as part of execution γ. Roughly speaking, γ starts with $\sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$ (see Construction 1) up to the point that $p_{k\%2}$ reports $x_{k\%2}^{in}$ for $X_{k\%2}$ to c_r^k (see Observation 1). Recall that only processes c_r^k and $p_{k\%2}$ take steps in $\sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$ (by Observation 1). Then, the events of β_{new} are executed to take the system in a configuration where x_0 and x_1 are visible. The assumption we made above to derive a contradiction, allows us to design β_{new} so that, c_r^k and $p_{k\%2}$ do not take any steps in it, and the values written by T_w are visible after β_{new} is applied starting from C_{k-1} (as well as from $RC(C_{k-1}, \sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k)))$. Afterwards, an execution γ_{new}, which is derived based on Construction 2, is applied from the resulting configuration. In γ_{new} only process $p_{(k-1)\%2}$ take steps (see Observation 2). Execution β_{new} has been designed so that $p_{(k-1)\%2}$ is "unaware" of $p_{k\%2}$'s decision on what to report to c_r^k as the current value of the object that $p_{k\%2}$ stores. So, $p_{(k-1)\%2}$ reports $x_{(k-1)\%2}$ for $X_{(k-1)\%2}$ as it does in Construction 2 (Observation 2). The construction of γ concludes with c_r^k taking steps until T_r responds. We argue (below) that c_r^k receives in γ the same message regarding the value of $X_{k\%2}$ as in $\gamma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$, so it returns $x_{k\%2}^{in}$ for $X_{k\%2}$ (by Observation 1). We also argue that c_r^k receives in γ the same message regarding the value of $X_{(k-1)\%2}$ as in γ_{new}, so it returns $x_{(k-1)\%2}$ for $X_{(k-1)\%2}$ (by Observation 2). This contradicts Lemma 1.

We first define β. For the base case (where $k = 1$), c_w invokes T_w starting from C_0 and executes solo (i.e., only c_w, p_0, p_1 take steps) until x_0 and x_1 are visible; since T_w has not yet been invoked in C_0, x_0 and x_1 are not visible in C_0. For the induction step, the induction hypothesis (claim 2) implies that in C_{k-1}, x_0 and x_1 are not visible. Again, we let c_w execute solo, starting from C_{k-1}, until x_0 and x_1 are visible (minimal progress implies that this will eventually happen). In either case, let C_v be the first configuration after C_{k-1} in which x_0 and x_1 are visible. Let β be the sequence of steps taken from C_{k-1} to C_v (all of them are by c_w and the servers).

In this and the next two paragraphs, we define β_{new} and show that it is legal from C_{k-1}. Let β'_p be the shortest prefix of β which contains all messages sent by c_w to $p_{(k-1)\%2}$, and let β'_s be the remaining suffix of β. Let β_p be the subsequence of β'_p in which all steps taken by $p_{k\%2}$ have been removed. Let β_s be the subsequence of β'_s containing only steps by $p_{(k-1)\%2}$. Let β_{new} be $\beta_p \cdot \beta_s$. Note that β_{new} does not contain any step by $p_{k\%2}$. Executions β, β_p and β_s are illustrated in Figure 3a. In the figure, symbols $i, 1-i \in \{0, 1\}$ refer to $k\%2$ and $(k-1)\%2$ respectively.

To show that β_{new} is legal from C_{k-1}, we first argue that β_p is legal from C_{k-1}. Because β'_p is a prefix of β, β'_p is legal from C_{k-1}. By assumption, $p_{k\%2}$ sends no message to $p_{(k-1)\%2}$, so if $p_{(k-1)\%2}$ receives any message from $p_{k\%2}$ in β'_p, then the message must have been sent before C_{k-1} (and must have been received after it), i.e., the message is not sent in β'_p. (For the base case, since no message is in transit in C_0, $p_{(k-1)\%2}$ receives no message from $p_{k\%2}$ in β'_p.) Moreover, by assumption, after α_{k-1}, $p_{k\%2}$ sends no message to c_w for which it holds that after c_w receives it, c_w sends a message to $p_{(k-1)\%2}$. Since, by definition, β'_p ends with a message sent by c_w to $p_{(k-1)\%2}$ (if β'_p is not empty), it follows that: for the base case, c_w receives no message from $p_{k\%2}$ in β'_p; for the induction step, if c_w receives any message from $p_{k\%2}$ in β'_p, then the message has been sent before C_{k-1}. Thus, β_p, which results from the removal of all steps taken by $p_{k\%2}$ from β'_p, is legal from C_{k-1}. Moreover, $RC(C_{k-1}, \beta'_p)$ and $RC(C_{k-1}, \beta_p)$ (i.e., the configurations that result when β'_p and β_p, respectively, are applied from C_{k-1}) are indistinguishable to $p_{(k-1)\%2}$ and c_w.

To complete the argument that β_{new} is legal from C_{k-1}, it remains to prove that β_s is legal from $RC(C_{k-1}, \beta_p)$. By definition, only $p_{(k-1)\%2}$ takes steps in β_s. Note that, because $RC(C_{k-1}, \beta'_p)$

and $RC(C_{k-1}, \beta_p)$ are indistinguishable to $p_{(k-1)\%2}$, by proving that β_s is legal from $RC(C_{k-1}, \beta'_p)$, it follows that β_s is legal from $RC(C_{k-1}, \beta_p)$. We next argue that β_s is indeed legal from configuration $RC(C_{k-1}, \beta'_p)$. By assumption, if $p_{(k-1)\%2}$ receives any message from $p_{k\%2}$ in β_s, then the message has been sent before C_{k-1} (i.e., the message has not been sent in β). For the base case, since no message is in transit in C_0, $p_{(k-1)\%2}$ receives no message from $p_{k\%2}$ in β_s. Recall that, by definition of β'_p, all messages from c_w to $p_{(k-1)\%2}$ are sent by the end of β'_p. Therefore, c_w does not send any message to $p_{(k-1)\%2}$ in β'_s. So, any message that $p_{(k-1)\%2}$ receives from c_w in β_s has been sent by the end of β'_p. Thus, β_s is legal from $RC(C_{k-1}, \beta'_p)$, and since $RC(C_{k-1}, \beta'_p)$ and $RC(C_{k-1}, \beta_p)$ are indistinguishable to $p_{(k-1)\%2}$, β_s is also legal from $RC(C_{k-1}, \beta_p)$. Therefore, $\beta_{new} = \beta_p \cdot \beta_s$ is legal from C_{k-1}.

From the arguments above, it also follows that $RC(C_{k-1}, \beta_p \cdot \beta_s)$ and $C_v = RC(C_{k-1}, \beta'_p \cdot \beta'_s)$ are indistinguishable to $p_{(k-1)\%2}$. Therefore, $RC(C_{k-1}, \beta_{new})$ and C_v are indistinguishable to $p_{(k-1)\%2}$.

We continue to construct γ. To do so, we use $\sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$ (Construction 1), β_{new}, and $\sigma_{new}(C_v, p_{k\%2}, c_r^k)$ (Construction 2). We also refer to configuration $C_{new}(C_v, p_{k\%2}, c_r^k)$ (Construction 2). Figure 3b illustrates the construction of γ, where symbols i, $1 - i \in \{0, 1\}$ refer to $k\%2$ and $(k-1)\%2$ respectively. (For simplicity, we omit $(C_{k-1}, p_{k\%2}, c_r^k)$ and $(C_v, p_{k\%2}, c_r^k)$ from the notations below.)

Recall that in γ, a client starts a read-only transaction from C_{k-1}. One server responds to the client first (i.e., σ_{old} is applied from C_{k-1}). Then the write-only transaction makes progress and the values written turn to be visible (specifically, β_{new} is applied after σ_{old}). Then the other server receives the request of the read-only transaction and responds to the client (specifically, σ_{new} is applied). Recall that (as we argue below) to one server, γ is indistinguishable from γ_{old} (i.e., the execution illustrated in Figure 3b is indistinguishable from that in Figure 2a) and thus the server returns an old value, while to the other server, γ is indistinguishable from γ_{new} (i.e., the execution illustrated in Figure 3b is indistinguishable from that in Figure 2b) and thus the other server returns a new value. This then leads to the contradiction.

We are now ready to formally define γ. Starting from C_{k-1}, the adversary applies $\sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k) \cdot \beta_{new} \cdot \sigma_{new}(C_v, p_{k\%2}, c_r^k))$ (we later prove that the application of these steps from C_{k-1} is legal). By Construction 1, in the last step of σ_{old}, $p_{k\%2}$ sends a message $mo'_{k\%2}$ to c_r^k. Similarly, by Construction 2, in the last step of σ_{new}, $p_{(k-1)\%2}$ sends a message $mn'_{(k-1)\%2}$ to c_r^k. The adversary next schedules the delivery of $mo'_{k\%2}$ and $mn'_{(k-1)\%2}$, and lets c_r^k take steps until T_r completes (this will happen because T_r is a fast transaction). This concludes the construction of γ.

We argue that γ is legal. By construction, only processes c_r^k and $p_{k\%2}$ take steps in σ_{old}. By Observation 1 (claim 2), C_{k-1} and $RC(C_{k-1}, \sigma_{old})$ are indistinguishable to c_w and $p_{(k-1)\%2}$. Since only c_w and $p_{(k-1)\%2}$ take steps in β_{new}, it follows that β_{new} is legal from $RC(C_{k-1}, \sigma_{old})$. Since the processes that take steps in σ_{old} and β_{new} are disjoint, $RC(C_{k-1}, \sigma_{old} \cdot \beta_{new})$ and $RC(C_{k-1}, \beta_{new} \cdot \sigma_{old})$ are indistinguishable to all processes. Recall that $RC(C_{k-1}, \beta_{new})$ and C_v are indistinguishable to $p_{(k-1)\%2}$. Since σ_{old} is composed of a sequence of steps in which only c_r^k and $p_{k\%2}$ take steps, $RC(C_v, \sigma_{old})$ and $C_{new}(C_v, p_{k\%2}, c_r^k)$ are indistinguishable to $p_{(k-1)\%2}$. It follows that $RC(C_{k-1}, \beta_{new} \cdot \sigma_{old})$ and $C_{new}(C_v, p_{k\%2}, c_r^k)$ are indistinguishable to $p_{(k-1)\%2}$. Because only $p_{(k-1)\%2}$ takes steps in σ_{new} and σ_{new} is legal from C_{new}, it follows that σ_{new} is legal from $RC(C_{k-1}, \sigma_{old} \cdot \beta_{new})$. Therefore, γ is legal.

We now focus on the values returned by c_r^k for T_r in γ. In γ, c_r^k executes only transaction T_r. Thus, c_r^k decides the response for T_r based solely on the values included in $mo'_{k\%2}$ (sent by $p_{k\%2}$ in σ_{old}) and $mn'_{(k-1)\%2}$ (sent by $p_{(k-1)\%2}$ in σ_{new}). For the base case, $mo'_{k\%2}$ is sent before c_w takes any step, so $mo'_{k\%2}$ contains neither x_0 nor x_1. For the induction step, since $mo'_{k\%2}$ is sent in γ_{old}, by Observation 1 and the one-value messages property, $mo'_{k\%2}$ contains neither x_0 nor x_1. Recall that $mn'_{(k-1)\%2}$ is sent in γ_{new}. By Observation 2 and the one-value messages property, $mn'_{(k-1)\%2}$ contains the value $x_{(k-1)\%2}$. Recall that (by assumption) c_r^k does not receive any other messages from p_0 and p_1. Thus, c_r^k receives for $X_{k\%2}$ just one value, namely, the same value it receives for it in γ_{old}. Similarly, it receives for $X_{(k-1)\%2}$ just one value, namely, the same value it receives for it in γ_{new}. It follows that in γ, the values that c_r^k returns are $x_{k\%2}^{in}$ for $X_{k\%2}$ and $x_{(k-1)\%2}$ for $X_{(k-1)\%2}$. This contradicts Lemma 1. (Note that γ is an execution utilized just for proving claim 1; for every $k > 1$, we build it from scratch to prove the induction step for k.)

Definition of α_k and C_k. We now define α_k and C_k. By claim 1, it follows that in any legal execution starting from C_{k-1} in which c_w executes solo, at least one of the following two statements hold: (1) $p_{k\%2}$ sends a message to $p_{(k-1)\%2}$; (2) $p_{k\%2}$ sends a message to c_w so that after c_w receives this message, c_w sends a message to $p_{(k-1)\%2}$. Let ms_k be the first message that satisfies any of the two statements above. We construct execution α'_k as follows. In α'_k, T_w executes solo starting from C_{k-1} until ms_k is sent[7]. Let $\alpha_k = \alpha_{k-1} \cdot \alpha'_k$, and let C_k be the configuration that results when α_k is applied from C_0.

Proof of claim 2. We use similar arguments as in the proof of claim 1. We assume that claim 2 does not hold, i.e., we assume that in C_k, x_i is visible for some $i \in \{0, 1\}$. To derive a contradiction, we construct an execution δ (similarly to that the construction of γ) and show that δ contradicts Lemma 1. We first define executions ρ and ρ_{new} in a way similar to β and β_{new} defined in the proof of claim 1. We finally define δ based on $\sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$, ρ_{new}, and $\sigma_{new}(C_k, p_{k\%2}, c_r^k)$, and we argue that in δ, c_r^k returns a response for T_r that contradicts Lemma 1.

We now present the details of the proof of claim 2. Assume that in C_k, x_i is visible for some $i \in \{0, 1\}$. We construct δ by utilizing executions $\sigma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$ and $\gamma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$ (Construction 1), as well as $\sigma_{new}(C_k, p_{k\%2}, c_r^k)$ and $\gamma_{new}(C_k, p_{k\%2}, c_r^k)$ (Construction 2). By Observation 1, c_r^k returns $x_{(k-1)\%2}^{in}$ for $X_{(k-1)\%2}$ and $x_{k\%2}^{in}$ for $X_{k\%2}$ for T_r in $\gamma_{old}(C_{k-1}, p_{k\%2}, c_r^k)$. Because by assumption, x_i is visible at C_k, Observation 2 implies that c_r^k returns $x_{(k-1)\%2}$ for $X_{(k-1)\%2}$ and $x_{k\%2}$ for $X_{k\%2}$ for T_r in $\gamma_{new}(C_k, p_{k\%2}, c_r^k)$.

To construct δ, we first define an execution ρ and some subsequences of it, and we study their properties. (The construction of ρ and its subsequences is similar to that of β and its subsequences. Yet, the reasoning of why the construction is legal is different.) Let ρ be

[7] Clearly, in α'_k, $p_{k\%2}$ must take at least one step. For the case where $k \geq 2$, we also require that in α'_k, in the first step which $p_{k\%2}$ takes, message ms_{k-1} is delivered at $p_{k\%2}$ (in order to comply with our model of finite message delay).

the sequence of steps which are taken from C_{k-1} to C_k, i.e., $\rho = \alpha'_k$. Let ρ'_p be the shortest prefix of ρ which contains all messages sent by c_w to $p_{(k-1)\%2}$, and let ρ'_s be the remaining suffix of ρ. Let ρ_p be the subsequence of ρ'_p in which all steps taken by $p_{k\%2}$ have been removed. Let ρ_s be the subsequence of ρ'_s containing only steps by $p_{(k-1)\%2}$. Let ρ_{new} be $\rho_p \cdot \rho_s$. We utilize ρ_{new} as part of our construction of δ below (as we did with β_{new} and γ). In the next two paragraphs, we argue that ρ_{new} is legal from C_{k-1}.

We first argue that ρ_p is legal from C_{k-1}. Because ρ'_p is a prefix of ρ, ρ'_p is legal from C_{k-1}. If $k = 1$ (base case), since no message is in transit in C_0, the definition of α_k implies that $p_{(k-1)\%2}$ receives no message from $p_{k\%2}$ in ρ'_p. For the induction step, the definition of α_k implies that if $p_{(k-1)\%2}$ receives any message from $p_{k\%2}$ in ρ'_p, then the message has been sent before C_{k-1}, i.e., the message is not sent in ρ'_p. Also, by definition of α'_k, $p_{k\%2}$ sends no message to c_w for which it holds that after its receipt, c_w sends a message to $p_{(k-1)\%2}$. By definition, ρ'_p ends with a message sent by c_w to $p_{(k-1)\%2}$ (if ρ'_p is not empty). It follows that: for the base case, c_w receives no message from $p_{k\%2}$ in ρ'_p; for the induction step, if c_w receives any message from $p_{k\%2}$ in ρ'_p, then the message has been sent before C_{k-1}. Thus, ρ_p, which results from the removal of all steps taken by $p_{k\%2}$ from ρ'_p, is legal from C_{k-1}. Also, $RC(C_{k-1}, \rho'_p)$ and $RC(C_{k-1}, \rho_p)$ are indistinguishable to $p_{(k-1)\%2}$ and c_w.

To complete the argument that ρ_{new} is legal from C_{k-1}, it remains to argue that ρ_s is legal from $RC(C_{k-1}, \rho_p)$. By definition, only $p_{(k-1)\%2}$ takes steps in ρ_s. note that, because $RC(C_{k-1}, \rho'_p)$ and $RC(C_{k-1}, \rho_p)$ are indistinguishable to $p_{(k-1)\%2}$, by proving that ρ_s is legal from $RC(C_{k-1}, \rho'_p)$, it follows that ρ_s is legal from $RC(C_{k-1}, \rho_p)$. We next argue that ρ_s is indeed legal from configuration $RC(C_{k-1}, \rho'_p)$. By the definition of α_k, if $p_{(k-1)\%2}$ receives any message from $p_{k\%2}$ in ρ_s, then the message has been sent before C_{k-1} (i.e., the message has not been sent in ρ). For the base case, since no message is in transit in C_0, the definition of α_k implies that $p_{(k-1)\%2}$ receives no message from $p_{k\%2}$ in ρ_s. Recall that by definition of ρ'_s, all messages from c_w to $p_{(k-1)\%2}$ are sent by the end of ρ'_p). Thus, ρ_s is legal from $RC(C_{k-1}, \rho'_p)$, and since $RC(C_{k-1}, \rho'_p)$ and $RC(C_{k-1}, \rho_p)$ are indistinguishable to $p_{(k-1)\%2}$, ρ_s is also legal from $RC(C_{k-1}, \rho_p)$. Therefore, $\rho_{new} = \rho_p \cdot \rho_s$ is legal from C_{k-1}.

From the arguments above, it also follows that $RC(C_{k-1}, \rho_p \cdot \rho_s)$ and $C_k = RC(C_{k-1}, \rho'_p \cdot \rho'_s)$ are indistinguishable to $p_{(k-1)\%2}$. Because $RC(C_{k-1}, \rho'_p)$ and $RC(C_{k-1}, \rho_p)$ are indistinguishable to $p_{(k-1)\%2}$, $RC(C_{k-1}, \rho_{new})$ and C_k are indistinguishable to $p_{(k-1)\%2}$.

We are now ready to formally define δ. To do so, we use executions $\sigma_{old}(C_{k-1}, p_{k\%2}, c^k_r)$ (Construction 1), $\sigma_{new}(C_k, p_{k\%2}, c^k_r)$ (Construction 2) and ρ_{new}. We also refer to $C_{new}(C_k, p_{k\%2}, c^k_r)$ (Construction 2). Starting from C_{k-1}, the adversary applies the step sequence $\sigma_{old}(C_{k-1}, p_{k\%2}, c^k_r) \cdot \rho_{new} \cdot \sigma_{new}(C_v, p_{k\%2}, c^k_r))$ (we later prove that the application of these steps from C_{k-1} is legal). For simplicity, we omit $(C_{k-1}, p_{k\%2}, c^k_r)$ and $(C_v, p_{k\%2}, c^k_r)$ from the notations below (thus abusing notations σ_{new} and C_{new} which were also used in the proof of claim 1.) By Construction 1, in the last step of σ_{old}, $p_{k\%2}$ sends a message $mo'_{k\%2}$ to c^k_r. Similarly, by Construction 2, in the last step of σ_{new}, $p_{(k-1)\%2}$ sends a message $mn'_{(k-1)\%2}$ to c^k_r. The adversary next schedules the delivery of $mo'_{k\%2}$ and $mn'_{(k-1)\%2}$, and lets c^k_r take steps until T_r completes (this will happen because T_r is a fast transaction). This concludes the construction of δ.

We now argue that δ is legal. By construction, only processes c^k_r and $p_{k\%2}$ take steps in σ_{old}. By Observation 1 (claim 2), C_{k-1} and $RC(C_{k-1}, \sigma_{old})$ are indistinguishable to c_w and $p_{(k-1)\%2}$. Since only c_w and $p_{(k-1)\%2}$ take steps in ρ_{new}, it follows that ρ_{new} is legal from $RC(C_{k-1}, \sigma_{old})$. Since the processes that take steps in σ_{old} and ρ_{new} are disjoint, $RC(C_{k-1}, \sigma_{old} \cdot \rho_{new})$ and $RC(C_{k-1}, \rho_{new} \cdot \sigma_{old})$ are indistinguishable to all processes. Recall that $RC(C_{k-1}, \rho_{new})$ and C_k are indistinguishable to $p_{(k-1)\%2}$. Since only c^k_r and $p_{k\%2}$ take steps in σ_{old}, $RC(C_k, \sigma_{old})$ and $C_{new}(C_k, p_{k\%2}, c^k_r)$ are indistinguishable to $p_{(k-1)\%2}$. It follows that $RC(C_{k-1}, \rho_{new} \cdot \sigma_{old})$ and $C_{new}(C_v, p_{k\%2}, c^k_r)$ are indistinguishable to $p_{(k-1)\%2}$. Because only $p_{(k-1)\%2}$ takes steps in σ_{new} and σ_{new} is legal from C_{new} (by definition), it follows that σ_{new} is legal from $RC(C_{k-1}, \sigma_{old} \cdot \rho_{new})$. Therefore, δ is legal.

We now focus on the values returned by c^k_r for T_r in δ. In δ, c^k_r executes only transaction T_r. Thus, c^k_r decides the response for T_r based solely on the values included in $mo'_{k\%2}$ (sent by $p_{k\%2}$ in σ_{old}) and $mn'_{(k-1)\%2}$ (sent by $p_{(k-1)\%2}$ in σ_{new}). For the base case, $mo'_{k\%2}$ is sent before c_w takes any step, so $mo'_{k\%2}$ contains neither x_0 nor x_1. For the induction step, since $mo'_{k\%2}$ is sent in σ_{old}, by Observation 1 and the one-value messages property, $mo'_{k\%2}$ contains neither x_0 nor x_1. Recall that $mn'_{(k-1)\%2}$ is sent in σ_{new}. By Observation 2 and the one-value messages property, $mn'_{(k-1)\%2}$ contains the value $x_{(k-1)\%2}$. Recall that (by assumption) c^k_r does not receive any other messages from p_0 and p_1. Thus, c^k_r receives for $X_{k\%2}$ just one value, namely, the same value it receives for it in γ_{old}. Similarly, it receives for $X_{(k-1)\%2}$ just one value, namely, the same value it receives for it in γ_{new}. It follows that in δ, the values that c^k_r returns are $x^{in}_{k\%2}$ for $X_{k\%2}$ and $x_{(k-1)\%2}$ for $X_{(k-1)\%2}$. This contradicts Lemma 1. □

3.4 The Limits of the Impossibility Result

Theorem 1 shows that multi-object write transactions (W) are incompatible with nonblocking (N), one-roundtrip (O) and one-value (V) read-only transactions. In this section, we investigate the limits of our impossibility result. We show that it is sufficient to relax any of these properties to obtain a distributed storage system that satisfies the rest. To this end, we describe possible designs that achieve combinations of three out of the four properties.

N + R + V. This combination supports fast read-only transactions and is implemented by COPS-SNOW [41]. When a client c writes a new value x_1 of object X_1, c piggybacks the information about its causal dependencies.Before making x_1 visible, the server p_1 storing X_1 contacts all servers that store objects listed in such dependency list. For each such object X, p_1 collects the identifiers of the read-only transactions that have read a value of X that is not the last written. Then p_1 enforces that x_1 is invisible to these read-only transactions. This prevents a read-only transaction from reading x_1 and then x_0, if in the meanwhile x'_0 has been created such that $x_0 <^c x'_0 <^c x_1$. Recall that COPS-SNOW does not ensure the W property, i.e., it does not support multi-object write transactions.

N + V + W. This design is implemented by Wren [55]. In this system, the servers periodically exchange information about the minimum timestamp among those of complete transactions. This *cutoff* timestamp is such that there does not exist any non-complete (or future) transaction with a lower timestamp. The cutoff timestamp is used to identify a snapshot of the data storage system from which a read-only transaction can read without blocking. A new object written by a client is assigned a timestamp higher than the cutoff timestamp, so as to reflect the causal dependencies of the object. Therefore, each client caches locally the values of the objects it writes, as long as their timestamps are smaller than the cutoff. This mechanism allows a client to read its own writes that are not included yet in the snapshot identified by the cutoff timestamp. Thus, each read-only transaction undergoes a first round of communication to get informed about the cutoff timestamp (we remark that this timestamp can be provided by any server) and then executes a second round of communication to actually read the objects.

N + R + W. Although we are not aware of any system that implements this design, we briefly discuss a modification of the COPS system [39] to achieve this combination of properties. COPS does not implement multi-object write transactions and implements read-only transactions that are nonblocking but may require two rounds of communication, each communicating just one value of the object to be read. We can augment COPS to achieve R and W as follows. Each write operation within a transaction must carry a) the values of the other objects written in the same transaction and b) information about *all* objects on which the transaction causally depends (including their values). This additional meta-data is stored with each written object. Hence, c executes a read-only transaction as follows. First, for each object o to read, c retrieves the value of o and the additional meta-data from the corresponding server. Then, once c has received a reply from each involved partition, c identifies, for each object, the last written value, which is returned to the application. This protocol is not efficient, as it requires to store and communicate a prohibitively big amount of data. It is an open problem whether a more efficient N+R+W protocol exists.

R + V + W. This design is implemented by RoCoCo-SNOW [41] and Spanner [19], which achieve strict serializability, and hence satisfy causal consistency. RoCoCo-SNOW implements a mechanism similar to COPS-SNOW, but assumes the *a priori* knowledge of the data accessed by transactions, which are executed as *stored procedures*. Spanner assumes tightly synchronized physical clocks and leverages the known bound on clock drift to order transactions. It is an open problem whether a R + V + W implementation exists that does not rely on such assumptions.

4 RELATED WORK

Existing systems. Table 1 characterizes existing systems from the point of view of the sub-properties of fast read-only transactions that they achieve, their support for multi-object write transactions, and their target consistency level. Consistently with our theorem, none among the systems that target the system model described in Section 2 implements multi-object write transactions and fast read-only transactions. Several systems achieve three out of the four properties we consider, and COPS-SNOW is the only one that implements fast read-only-transactions while complying with our system model. Our theorem implies that the design of these systems cannot be improved with respect to the properties we consider.

System	Fast ROT			WTX	Consistency
	R	V	N		
RAMP [12]	≤ 2	≤ 2	yes	yes	Read Atomicity [12]
COPS [39]	≤ 2	≤ 2	yes	no	Causal Consistency [2]
Orbe [25]	2	1	no	no	Causal Consistency
GentleRain [26]	2	1	no	no	Causal Consistency
ChainReaction [4]	≥ 1	≥ 1	no	no	Causal Consistency
POCC [32]	2	1	no	no	Causal Consistency
Contrarian [23]	2	1	yes	no	Causal Consistency
COPS-SNOW [41]	1	1	yes	no	Causal Consistency
Eiger [40]	≤ 3	≤ 2	yes	yes	Causal Consistency
Wren [55]	2	1	yes	yes	Causal Consistency
PaRiS [56]	2	1	yes	yes	Causal Consistency
SwiftCloud† [63]	1	1	yes	yes	Causal Consistency
Cure [3]	2	1	no	yes	Causal Consistency
Yesquel [1]	1	1	no	yes	Snapshot Isolation [14]
Occult [43]	≥ 1	≥ 1	yes	yes	Per Client Parallel SI [43]
Granola [20]	2	1	yes	yes	Serializability [15]
TAPIR [64]	≤ 2	1	yes	yes	Serializability
Eiger-PS† [41]	1	1	yes	yes	PO-Serializability [41]
Spanner† [19]	1	1	no	yes	Strict Serializability [49]
DrTM [61]	≥ 1	≥ 1	no	yes	Strict Serializability
RoCoCo [44]	≥ 1	≥ 1	no	yes	Strict Serializability
RoCoCo-SNOW [41]	1	1	no	yes	Strict Serializability
Calvin [59]	2	1	no	yes	Strict Serializability

Table 1: Characterization of existing systems. Systems with a † rely on a different system model from the one we target.

SwiftCloud and Eiger-PS implement fast read-only transactions and support multi-object write transactions, but assume a system model that differs from the one we target. Although they eventually complete all writes, the values they write may be invisible to some clients for an indefinitely long time. Hence, read-only transactions may see very old values of some objects, even the initial ones. To improve the freshness of the data seen by the clients, servers can communicate with clients out of the scope of transactional operations. This requires that the servers maintain a view of the connected clients. Typically, there are far more clients than servers, so this design choice results in reduced performance and scalability and is avoided by state-of-the-art data platforms. Furthermore, SwiftCloud assumes only a single partition that stores the whole data set (potentially fully replicated across multiple sites).

Impossibility results. Existing impossibility results on storage systems typically rely on stronger consistency or progress properties. Brewer [16] conjectured the CAP theorem, according to which no implementation guarantees *consistency*, *availability*, and *network partition tolerance*. Gilbert and Lynch [28] formalized and proved this conjecture. Specifically, they formalized consistency by using the notion of *atomic* objects [34] (i.e., by assuming *linearizability* [29], which is stronger than causal consistency). Roohitavaf *et al.* [53] considered a replicated storage system implemented using *data centers* (i.e., clusters of servers), and a model in which any value written is *immediately* visible to the reads initiated in the same data center. They proved that it is impossible to ensure causal consistency, availability and network partition tolerance across data centers. Their proof (as well as the proof of the CAP Theorem) rely on message losses, whereas in our model no message can be lost.

Mahajan *et al.* [50] proved that no implementation guarantees one-way convergence (a progress condition stating that if processes communicate appropriately, then they eventually converge on the values they read for objects), availability, and any consistency stronger than real time causal consistency [50] assuming that messages may be lost. In their model, communication may occur among any pair of processes. On the contrary, in our model, communication cannot occur directly between clients, the progress property we assume is simpler (and decoupled from the underlying communication), and no message may be lost.

Variants of causal consistency motivated by replicated systems have been presented in [8, 62]. Their definitions are based on the events that are executed at the servers (and not on the histories of operations executed in the transactions issued by the clients). Attiya *et al.* [8] proved that a (non-transactional) replicated storage system implementing *multi-valued registers* (i.e., registers for which a read returns the set of values written by conflicting writes) cannot satisfy any consistency strictly stronger than observable causal consistency. Xiang and Vaidya [62] defined the notion of *replica-centric causal consistency*, and they proved that (non-transactional) replicated distributed storage systems ensuring this consistency property have to track writes. These works are in different avenues than our work and focus on other models than that in our paper.

Lu *et al.* [41] proved the SNOW theorem, which shows that no fully-functional distributed transactional system can support fast *strictly serializable* read-only transactions. Lu *et al.* also showed that any fully-functional distributed transactional system that achieves a consistency level weaker than or equal to process-ordered serializability [41] (hence including causal consistency) can support fast read-only transactions. Tomsic et al. [60] further showed that implementing fast read-only transactions with an order-preserving consistency level (as is the case for causal consistency) is possible only by allowing read-only transactions to read possibly stale values of the objects being accessed. These results may seem at odds with our impossibility result. However, these results rest on very weak assumptions on the progress guarantees of write operations. Although they assume that all writes eventually complete, the values they write may be invisible to clients for an indefinitely long time. Such a weak assumption allows the design of trivial algorithms in which read-only transactions can return arbitrarily old values –even the initial ones– for the objects they read.

Recently, Didona *et. al* [23] showed a lower bound on the number of bits that must be communicated in order to support fast causally consistent read-only transactions in distributed storage systems. On the contrary, this paper focuses on the *design* implications of fast read transactions in distributed transactional such systems, showing that they are incompatible with multi-object write transactions.

Since the introduction of causal consistency by Ahamad *et al.* [2] for a shared memory system, other versions of causal consistency has been studied [39, 52]. Our result holds if we replace our definition of causal consistency with those provided in these papers.

5 CONCLUSION

We present an impossibility result that establishes a fundamental trade-off in the design of distributed transactional storage systems: fast read transactions cannot be achieved by fully-functional transactional storage systems. The design of such systems must either sacrifice fast read transactions, or must settle for reduced functionality, i.e., support only single-object write transactions.

Unlike most previous work on distributed transactional systems, which target strong consistency, our result assumes only causal consistency. This broadens the scope of our result which applies also to systems that implement any consistency level stronger than causal consistency, or a hybrid consistency level that includes causal consistency. Proving our result under such weak consistency model is nontrivial and required us to devise a complex proof.

Our result sheds light on the design choices of state-of-the-art distributed transactional storage systems, and is useful for the architects of such systems because it identifies impossible designs.

Our result also opens several interesting research questions, such as investigating which is the weakest consistency condition for which our impossibility result holds. In addition, it is interesting to further investigate the design of systems that provide some of the combinations of the studied properties, as discussed in Section 3.4.

ACKNOWLEDGMENTS

This research has been supported by the European Union's H2020-FETHPC-2016 programme under grant agreement No 754337 (project EuroExa), by the European ERC Grant 339539-AOC, and by an Eco-Cloud post-doctoral research fellowship. Part of this work has been done while P. Fatourou was working as a visiting professor at EPFL.

REFERENCES

[1] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. 2015. Yesquel: Scalable Sql Storage for Web Applications. In *SOSP*.
[2] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49.
[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. PreguiÃČÂğa, and M. Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS*.
[4] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *EuroSys*.
[5] Masoud Saeida Ardekani and Douglas B. Terry. 2014. A Self-Configurable Geo-Replicated Cloud Storage System. In *ATC*.
[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*.
[7] Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of Highly-Available Eventually-Consistent Data Stores. In *PODC*.
[8] H. Attiya, F. Ellen, and A. Morrison. 2017. Limitations of Highly-Available Eventually-Consistent Data Stores. *IEEE TPDS* 28, 1 (Jan 2017), 141–155.
[9] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc.
[10] Microsoft Azure. 2018. CosmosDB. Online. https://azure.microsoft.com/en-us/services/cosmos-db/.
[11] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *VLDB Endow.* 7, 3 (Nov. 2013).
[12] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD*.
[13] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *EuroSys*.
[14] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*.
[15] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing.
[16] Eric A. Brewer. 2010. Towards robust distributed systems. In *Talk at PODC*.
[17] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC*.
[18] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana

Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288.
[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM TOCS* 31, 3, Article 8 (Aug. 2013), 22 pages.
[20] James Cowling and Barbara Liskov. 2012. Granola: Low-overhead Distributed Transaction Coordination. In *ATC*.
[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*.
[22] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2019. Distributed Transactional Systems Cannot Be Fast. *CoRR* abs/1903.09106 (2019). arXiv:1903.09106 http://arxiv.org/abs/1903.09106
[23] Diego Didona, Guerraoui Rachid, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe?. In *VLDB*.
[24] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *NSDI*.
[25] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *SOCC*.
[26] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *SOCC*.
[27] Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. 2015. *Consistency for Transactional Memory Computing.* Springer International Publishing, 3–31.
[28] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59.
[29] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.
[30] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*.
[31] Kishori M. Konwar, Wyatt Lloyd, Haonan Lu, and Nancy A. Lynch. 2018. The SNOW Theorem Revisited. *CoRR* abs/1811.10577 (2018). http://arxiv.org/abs/1811.10577
[32] Spirovska Kristina, Didona Diego, and Zwaenepoel Willy. 2017. Optimistic Causal Consistency for Geo-Replicated Key-Value Stores. In *ICDCS*.
[33] Cockroach Labs. 2017. CockroachDB. Online. https://www.cockroachlabs.com.
[34] Leslie Lamport. 1986. On interprocess communication. *Distributed Computing* 1, 2 (01 Jun 1986), 77–85.
[35] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *ATC*.
[36] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*.
[37] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *ISCA*.
[38] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *NSDI*.
[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *SOSP*.
[40] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*.
[41] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *OSDI*.
[42] Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers.
[43] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *NSDI*.
[44] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *OSDI*.
[45] MySQL. 2018. MySQL Cluster. Online. https://www.mysql.com/.
[46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *NSDI*.
[47] Shadi A. Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H. Campbell. 2016. Ambry: LinkedIn's Scalable Geo-Distributed Object Store. In *SIGMOD*.
[48] Oracle. 2018. Coherence. Online. http://www.oracle.com/technetwork/middleware/coherence/overview/index.html.
[49] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *JACM* 26, 4 (Oct. 1979), 631–653.
[50] Lorenzo Alvisi Prince Mahajan and Mike Dahlin. 2011. *Consistency, Availability, and Convergence.* Technical Report UTCS TR-11-22. Department of Computer Science, The University of Texas at Austin.
[51] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *SIGMOD*.
[52] M. Raynal, G. Thia-Kime, and M. Ahamad. 1997. From serializable to causal transactions for collaborative applications. In *EUROMICRO*.
[53] M. Roohitavaf, M. Demirbas, and S. Kulkarni. 2017. CausalSpartan: Causal Consistency for Distributed Data Stores Using Hybrid Logical Clocks. In *SRDS*.
[54] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *SIGMOD*.
[55] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2018. Wren: Nonblocking Reads in a Partitioned Transactional Causally Consistent Data Store. In *DSN*.
[56] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *ICDCS*.
[57] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.
[58] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *SOSP*.
[59] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*.
[60] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. 2018. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware*.
[61] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing Using RDMA and HTM. In *SOSP*.
[62] Zhuolun Xiang and Nitin H. Vaidya. 2017. Lower Bounds and Algorithm for Partially Replicated Causally Consistent Shared Memory. *CoRR* abs/1703.05424 (2017).
[63] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Middleware*.
[64] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *SOSP*.