The Impossibility of Fast Transactions

Karolos Antoniadis
EPFL
karolos.antoniadis@epfl.ch

Diego Didona
IBM Research - Zurich
ddi@zurich.ibm.com

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Willy Zwaenepoel
University of Sydney
willy.zwaenepoel@sydney.edu.au

Abstract—**We prove that transactions cannot be fast in an asynchronous fault-tolerant system. Our result holds in any system where we require transactions to ensure monotonic writes, or any stronger consistency model, such as, causal consistency. Thus, our result unveils an important, and so far unknown, limitation of fast transactions: they are impossible if we want to tolerate the failure of even one server.**

I. Introduction

The surge of cloud computing and big data has led to the design of large-scale and highly available online services. A fundamental component of large-scale online services is a distributed data store [1]–[3]. Naturally, the demand for highly available online services translates to the demand for highly available data stores.

The CAP theorem [4], [5] states that a distributed system has to choose between availability and strong consistency during a network partition. In practice, networks are not reliable [6], and thus we can never eliminate the possibility of network partitions. For this reason, highly available data stores choose to sacrifice strong consistency in favor of availability [1]–[3], [7]. A number of well-known data stores support eventual consistency [1], [3], [7]. Eventual consistency [8] simply states that if we reach a quiescent state where no updates are taking place (i.e., no writes are issued by the clients), eventually all the servers contain non-conflicting data.[1] Recent work has shown that the strongest consistency model that can be achieved in the presence of network partitions is causal consistency [9], [10]. As a result, causal consistency has recently gained attention in academia [11]–[17], as well as in industry, where most notably, the MongoDB [3] data store supports causal consistency.

Typically, the interface of a data store is a read-write interface [2] on a set of objects, where the objects can be identified by what is called a key. To handle the enormous amount of data, data stores partition[2] the data (e.g., based on a key) across multiple servers. To avoid loss of data, these systems replicate the data to multiple servers. To remain highly available and reduce the latency of client operations, data stores are replicated across multiple geographically separated data centers. At a high-level, the client issues a request on the data store by identifying the object (e.g., by providing the key) the client wants to access. Subsequently, the server responds back to the client with the desired data. A number of data stores [3], [12]–[14], [18]–[20] augment their interface by providing transactions. Transactions operate on multiple objects at once and substantially aid the programmer's job. Data stores being extensively used for read-heavy workloads, aim to optimize read-only transactions since they are the most frequent in practice [21]. It is thus natural to seek implementations for read-only transactions that are as fast as possible.

Lu et al. [13] provide a first informal description of what it means for a read-only transaction to be *fast*, or as they call it, *latency-optimal*. Their definition, captures the fact that a read-only transaction is one round-trip, non-blocking, and one-version. One round-trip means that a client does not contact a server more than once during a transaction. Non-blocking [13] states that servers should not communicate with each other before responding to the client. Finally, one-version asks that a server only sends one value for each read object. In the same work, Lu et al. prove that fast read-only transactions are possible by presenting COPS-SNOW, a causally-consistent data store that provides fast read-only transactions. Because of the critical importance of fast read-only transactions for industrial data stores, fast read-only transactions have received much attention of late [12], [13], [22]–[24]. However, all this research [12], [13], [22]–[24] targets the ideal case where servers never fail.

In practice, distributed systems are deployed in settings where failures are the norm. Distributed systems ought to be fault-tolerant. Fault-tolerance is usually achieved by means of replication or logging. However, replication and logging are synchronous (i.e., blocking) operations. As expected, achieving fault-tolerance has a cost on the performance of transactions that write to objects. Write transactions cannot be fast since they are blocking: write transactions have to replicate data before committing. It is therefore natural that the description of fast transactions [13] only refers to read-only transactions.

In this paper we prove that, surprisingly, read-only transactions cannot be fast either. Fast transactions in general are impossible in a system that aims to be fault-tolerant. To show that read-only transactions cannot be fast, we examine the concept of the visibility of transactions, and investigate whether fast read-only transactions can be invisible. Transactions are said to be invisible if they do not modify the state of the servers with which they communicate. Intuitively, visible read-only transactions modify the state of the server they are operating

[1] In contrast to its name, eventual consistency is a liveness, rather than a safety property.

[2] In this context, we use the term "partition" in the sense of sharding.

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00120

on. Thus, visible read-only transactions are blocking, since the modification on the server has to be performed in a fault-tolerant way (e.g., by replicating the modification to other servers). This means, that in a fault-tolerant system, read-only transactions can only be fast if they are invisible.

We prove that in an asynchronous system, if we require transactions to ensure monotonic writes, a minimal level of consistency that is weaker than causal consistency, then read-only transactions cannot be both fast and invisible. In fact, our proof holds for a weaker definition of fast transactions, one where a server can send a bounded number of versions (instead of just one) back to the client. In this sense, we prove a more general result. Our result sheds some light on a so far unexplored limitation of fast transactions: they are impossible if we want to tolerate the failure of even one server. Furthermore, we prove that if a server can send an unbounded number of versions back to the client, then transactions can be non-blocking and one round-trip. To prove this, we devise a new data store algorithm that we believe is interesting in its own right, called `ubvStore`, that provides non-blocking and one round-trip transactions.

The practical implications of our theoretical results are threefold. First, similar to the CAP theorem [4], [5], we demonstrate a new trade-off for data stores: either fast transactions or fault-tolerance can be achieved, even with the weak consistency model of monotonic writes. Second, our results allow designers to avoid chasing impossible designs. Third, we show that fault-tolerance should be a first-class concern when proposing new theoretical properties, in order for the properties to have practical utility.

To prove our results, we devised a new formal framework that is general enough to capture any data store, while at the same time the framework is able to precisely capture notions such as bounded-version, non-blocking, etc., a challenging endeavor. As far as we know, our formalism is the first that precisely captures the notion of fast read-only transactions. We consider the framework as a contribution on its own.

Roadmap. The rest of this paper is organized as follows. We present our framework in Section II. In Section III, we prove the impossibility of fast transactions and we discuss the ramifications of our result. Then, in Section IV, we describe the `ubvStore` algorithm. Due to space constraints, we only present the main ideas behind `ubvStore` and present the full algorithm and its proof of correctness in the extended version of this paper [25]. Finally, in Section V, we discuss related work before concluding.

II. Model

We consider an asynchronous model that captures the notion of a data store that supports write operations on single objects, as well as read-only transactions that operate on groups of objects. Our model does not support transactions that write to objects, hence our impossibility result is stronger. Since we only consider read-only transactions, whenever we refer to a *transaction*, we refer to a read-only transaction. We distinguish between servers and clients in our system and clearly define the ways they can communicate and the exact type of messages a client can send to the server. Nevertheless, the model provides great flexibility on how the processes (i.e., clients and servers) can communicate.

We consider a *data store* as a message-passing system with servers and clients that communicate. Servers store objects that the clients can read or write. Specifically, a data store is a tuple $(\mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{V}, \mathcal{M}, dec)$ where $\mathcal{S}, \mathcal{C}, \mathcal{O}, \mathcal{V}$, and $\mathcal{M}$ are sets and dec is a function, as described below. We consider that a data store consists of n servers contained in a set $\mathcal{S} = \{s_1, \dots, s_n\}$, as well as a finite set $\mathcal{C} = \{c_1, c_2, \dots, c_m\}$ of m clients. We consider that both servers and clients are deterministic. Clients with servers, as well as servers with servers communicate by exchanging messages, where messages can take arbitrary time to be delivered but eventually are delivered (i.e., no message is lost). We assume that clients cannot communicate with each other. Additionally, we consider a set $\mathcal{O} = \{o_1, \dots, o_l\}$ of l objects and an infinite set $\mathcal{V} = \{v_1, v_2, \dots\} \cup \{\perp\}$ of finite values that the objects can take. Value $\perp$ corresponds to the initial value of each object. No write operation can write $\perp$ value to an object. Note that depending on the context, we refer interchangeably to a value as a *version*. We also consider an infinite set of messages $\mathcal{M}$, where each message $m \in \mathcal{M}$ is created over some alphabet. All the infinite sets we consider are countable. Finally, we consider the decoding function $dec : \mathcal{M} \to 2^{\mathcal{V}}$, that given a message m returns a set of values that are encoded in m. We use function dec to bound the number of values a client can utilize from a given message.

We introduce some notation that we use throughout the paper. For a set S we define $S^{\leq k} = S^1 \cup S^2 \cup S^3 \cup \dots \cup S^k$, where $S^1 = S$ and for $i > 1$ $S^i = S \times S^{i-1}$. For a tuple $v = (v_1, v_2, \dots, v_g)$ we denote with v_i the i-th element of v, e.g., $(3, 8, 1)_2$ is 8. Finally, given a sequence of elements $\alpha = a_1, a_2, \dots$, we denote with $a_i \in \alpha$ that a_i appears in α.

Server. We model a *server* as a state machine $s = (\Sigma, \sigma_0, E, \Delta, obj)$ where Σ is the set of possible states, $\sigma_0 \in \Sigma$ is the initial state and E is the set of possible events. $\Delta : \Sigma \times E \to \Sigma$ is a partial function that captures the possible state transitions a server can take based on a given event. The set $obj \subseteq \mathcal{O}$ is the set of objects that the server handles. For a server s, we denote with $s.\Sigma$, $s.\sigma_0$, $s.E$, $s.\Delta$, and $s.obj$ the set of states Σ of server s, the set of events E of server s, etc. We say that a server s *serves* objects $s.obj$ or a server *serves* an object o where $o \in s.obj$. We consider a system where all the objects are served by some server, hence $\bigcup_{i=1}^{n} s_i.obj = \mathcal{O}$, and additionally we assume that every object is being served by a single server, hence $\forall s, s' \in \mathcal{S}$ with $s \neq s'$ $s.obj \cap s'.obj = \emptyset$.

In what follows, we refer to either a server or a client as a *process*. For each server $s \in \mathcal{S}$, the set of events is defined as $s.E = \{send(m, p), receive(m, p) : m \in \mathcal{M}, p \in \mathcal{S} \cup \mathcal{C}\}$. Specifically, a $send(m, p)$ event corresponds to server s transmitting message m to process p. Event $receive(m, p)$ corresponds to server s receiving message m from process p.

Client. We model a *client* as a state machine $c = (\Sigma, \sigma_0, E, \Delta, \rho)$ where Σ, σ_0, and Δ are defined in the same way as of a server. Function ρ captures the notion of the exact values a client reads for a transaction. We consider an infinite set $\mathcal{T} = \{t_1, t_2, \dots\}$ of finite transactional identifiers. Then, function $\rho : \Sigma \times \mathcal{T} \to \mathcal{V}^{\leq l}$ (note that $|\mathcal{O}| = l$) is given a state $\sigma \in \Sigma$ and a transactional identifier and returns an arbitrary number of values. Function ρ is used to define the monotonic writes consistency property (see later on). Additionally, the set of events E for a client is different than that of a server, since for a client c we restrict the events c can take (i.e., the messages a client can send and receive).

A client can either perform a write operation on a single object, or a transaction on a set of objects that are served by more than one server. To clearly capture the notion of a transaction in our model, we consider that a client splits a transaction into multiple read operations where each read operation is destined to a different server with the objects to be read. Specifically, a client can only issue two kinds of operations, a $read$ and a $write$ operation. In what follows, we first describe the exact operations a client can issue. We then describe what kind of messages a client c can send and receive (i.e., events a client can take) based on the operations c performs. A $read(O_s, t, d)$ operation reads ℓ objects defined in ℓ-tuple $O_s \in \mathcal{O}^\ell$, and $d \in \mathcal{M}$ as part of some transaction with identifier $t \in \mathcal{T}$. Note that a read operation reads from distinct objects, thus for a read $read(O_s, t, d)$ where $O_s = (o_1, o_2, \dots, o_\ell)$ is an ℓ-tuple, $\forall i, j \in \{1, \dots, \ell\}$ with $i \neq j$, it is the case that $o_i \neq o_j$. A $read(O_s, t, d)$ operation is always part of a *transaction*. Naturally, a client can perform a transaction on objects that reside on different servers. In such a case, a client sends two read operations with the same transactional identifier to two different servers. For example, assume we have two servers $s_1, s_2 \in \mathcal{S}$ with $s_1.obj = \{o_1\}$, $s_2.obj = \{o_2\}$, and a client $c \in \mathcal{C}$ wants to perform a transaction that reads both objects o_1 and o_2. Then, client c has to issue a $read((o_1), t_{id}, d_1)$ operation to server s_1 and a $read((o_2), t_{id}, d_2)$ to server s_2.

A $write(o, v, d)$ operation writes value v to single-object o where $o \in \mathcal{O}, v \in \mathcal{V} \setminus \{\perp\}$, and $d \in \mathcal{M}$. Note that both the read and write operations take as a parameter a message d. This message is not necessarily bounded (since a message can be of any size) and can contain additional information the client might want to include in its operation. We specifically allow clients to send any message d in order to have a model as general as possible. This way, we do not restrict the possible data stores the model expresses.

Client responses. The response to a $read(O_r, t, d)$ operation with $|O_r| = r$ is $res(x)$ where $x \in \mathcal{O}^r \times \mathcal{M}$. Specifically, $res(x) = (O_r, m)$ where $m \in \mathcal{M}$. In other words, the response to a $read$ reading r objects is a pair of one tuple that contains the to-be-read r objects and the message m containing the values for the r objects. Naturally, a response to a single-object $read(o, t, d)$ is $res(x)$ where $x = (o, m)$ and $m \in \mathcal{M}$. Note that a message m can contain multiple values (i.e., versions) for a specific object. We present later how to use dec to restrict the possible values a client can retrieve from a message.

Similarly to a read operation, we consider that a response to a $write$ operation is $res(d)$ where $d \in \mathcal{M}$. Note that we can get a response $res(d)$ with $d \in \mathcal{M}$ only in response to a write operation.

A client can issue a transaction that consists of multiple read operations. The responses from the read operations are used to extract the values the transaction reads using ρ.

Client events. We describe the set of all messages the client can send or receive. The set of messages the client can send is $\mathcal{M}_s = \{m : m = read(\mathcal{O}_r, t_{id}, d)$ and $\mathcal{O}_r \in \mathcal{O}^{\leq l}, t_{id} \in \mathcal{T}, d \in \mathcal{M}\} \cup \{m : m = write(o, v, d)$ and $o \in \mathcal{O}, v \in \mathcal{V} \setminus \{\perp\}, d \in \mathcal{M}\}$. The set of messages the client can receive is $\mathcal{M}_r = \{m : m = res(x)$ and $x \in \mathcal{O}^{\leq l} \times \mathcal{M}\} \cup \{m : m = res(d)$ and $d \in \mathcal{M}\}$. The set of possible events a client $c \in \mathcal{C}$ can take is $c.E = \{send(m_s, p) : m_s \in \mathcal{M}_s, p \in \mathcal{S}\} \cup \{receive(m_r, p) : m_r \in \mathcal{M}_r, p \in \mathcal{S}\}$. Finally, note that clients cannot communicate with each other but only with servers, a natural assumption [13], [14], [20]. It might seem that a client can issue a read or a write operation for an object o to a server s that does not serve o. We restrict these cases later, when we define what a well-formed execution is.

Execution. We say that an event e is *enabled* in state σ if $\Delta(\sigma, e)$ is defined. An *execution* is a (possibly infinite) sequence of events occurring at the servers and the clients. A sequence of events occurring at a process p (i.e., p is either a server or a client) is *well-formed* if there is a sequence of states, $\sigma_1, \sigma_2, \dots$ such that $\sigma_i = p.\Delta(\sigma_{i-1}, e_i)$ for all $2 \leq i \leq$ (the length of the sequence). An execution has *correct issues* of operations if every $read(O_r, t, d)$ with $O_r = (o_1, \dots, o_\ell)$ that a client issues is destined to a server s where $\forall i, 1 \leq i \leq \ell, o_i \in s.obj$, as well as every $write(o, v, d)$ operation is destined to a server s where $o \in s.obj$. We assume that clients have some initial knowledge on which server contains which objects, a reasonable assumption in practice since such information could be stored in the initial state of each client.

We say that an event e is a *client write request* if e corresponds to the $send$ event of a client for a write operation. We say that an event e is a *client read request* if e corresponds to the $send$ event of a client for a read operation. For example, event $send(read(O_r, t, d), s)$ taken by some client is a client read request, while event $send(write(o, v, d), s)$ is a client write request. We say that an event e is a *client request* if e is a client read or write request. Similarly, we say that an event e is a *client read response* if e corresponds to the receipt event of a client with $res(x)$ and $x \notin \mathcal{M}$ (i.e., $e = receive(res(O_r, m), c)$). We call an event e a *client write response* if e corresponds to the receipt event of client with a $res(d)$ event where $d \in \mathcal{M}$. We say that an event e is a *client response* if e is a client read or write response. For brevity, we also use the notation $e = read(O_s, t, d)$, $e = write(o, v, d)$, or $e = res(x)$, when it is clear from the context whether event e is being sent or received by a client or by a server.

For a given client request e we define $obj(e)$ to be the tuple of objects e is operating on. For example, if e is a $read((o_5, o_8), t, d)$, then $obj(e) = (o_5, o_8)$. Similarly, for a client read request e that is associated with a transaction, $tx(e)$ provides the transactional identifier associated with e. Again, if an event e is taken by a client, we denote with $cl(e) \in \mathcal{C}$ the client that took e. For every process $p \in \mathcal{C} \cup \mathcal{S}$, given an execution α, we define the *process execution* $\alpha|_p$ to be the subsequence of α that contains all the events of α taken by process p. Given an execution α, we define the *read execution* $\alpha|_{read}$ to be the subsequence of α containing only client read requests and client read responses. Similarly, we define execution $\alpha|_{write}$ to be the subsequence of α containing only client write requests and client write responses.

Valid responses. An execution α has *no-thin-air responses*, if for every client $c \in \mathcal{C}$, for every client response event $e' = res(x)$ in $\alpha|_c$, there is a client request event e that precedes e' in $\alpha|_c$ such that e is either a $write$ event if $x \in \mathcal{M}$, or e is a $read(O_s, t, d)$ event if $x = (O_s, m)$ with $O_s \in \mathcal{O}^{\leq l}$ and $m \in \mathcal{M}$. As the name suggests, no-thin-air responses captures the notion that client responses are not created out of thin-air (i.e., a client request should trigger the response).

An execution α has *written-values responses*, if for every client $c \in \mathcal{C}$, for every client read response event $e' = res(x)$ with $x = (O_s, m)$, then for every $v \in dec(m)$, there should be an object $o \in O_s$ such that there is a client write request $e = write(o, v, d)$ that appears before e' in α. In other words, a server s can only send values that were at some point written to the objects the client is reading.

An execution α has *valid responses* if α has no-thin-air and written-values responses.

Sequential clients. Clients can issue reads as part of the same transaction to objects belonging to different servers. For example, a client might issue a transactional $read((o_1, o_2), t, d)$ to a server s_1 and another read $read((o_4, o_5), t, d)$ to a server s_2. Clients are said to be *sequential*. This means that a client can issue a write or a read operation only if the client has received responses to all its previous requests. Furthermore, a client c can issue a read with a transactional identifier t if c has received responses to a previous write request, as well as to all transactional reads of a transaction t' with $t' \neq t$. In other words, a client c can issue parallel read operations for the same transaction, but c has to wait for a response to its previous operations before issuing a write or a new transaction.

Formally, we say that an execution α has *sequential clients* if for every client $c \in \mathcal{C}$, for every client request $e \in \alpha|_c$, where e is not the last event in $e \in \alpha|_c$, the following holds:

- if $e = read(O_r, t, d)$, then the event e' that immediately succeeds e in $\alpha|_c$ has $tx(e') = tx(e)$ or $e' = res(x)$ with $x = (O_{r'}, m)$ with $m \in \mathcal{M}$ ($O_{r'}$ is not necessarily equal to O_r);
- if $e = write(o, v, d)$, then the event e' that immediately succeeds e is $res(d)$ with $d \in \mathcal{M}$.

Valid values. Next, we provide auxiliary definitions that help us capture the notion of a bounded-version data store.

Definition 1 (Corresponding event)**.** *Given an execution α that has no-thin-air responses, consider a client response $e' = res(x)$ where $e' \in \alpha$. Event e' is received by client $c \in \mathcal{C}$ in response to client's c request e. We say that event e' has e as its* corresponding event*, or that response e' has e as its* corresponding request*. Conversely, request e has e' as its* corresponding client response*.*

Given a client request e and its corresponding client response e' in an execution α, we denote e's corresponding client response with $cor(e)$. We say that a client request e is *completed* in an execution α if $cor(e) \in \alpha$. Note that a transaction can be split into many client read requests and the completion of some of these requests does not imply that the transaction has completed.

We say that a client *read response e is associated with a transaction t* if e's corresponding read request e' has $tx(e') = t$. A transaction t is *completed* in an execution α if there is a read request event $e \in \alpha$ with $tx(e) = t$ and $cl(e) = c$ and there is a read request event e' that succeeds e in $\alpha|_c$ such that $tx(e') \neq t$. Given an execution α, we define as $comp(\alpha) \subseteq \mathcal{T}$ the set of all completed transactions in α.

Definition 2 (Last state of a transaction)**.** *Consider an execution α and a transaction $t \in comp(\alpha)$ issued by a client c, we denote with $\sigma_{last}(t, \alpha)$ the last state of client c that was part of transaction t. $\sigma_{last}(t, \alpha)$ corresponds to the state immediately after the last event associated with t took place in α by c.*

The following definition helps us define correctness in an execution on what a transaction reads. For this, we need to know what values are read by a transaction.

Definition 3 (Values of a transaction)**.** *Given an execution α, we say a transaction $t \in comp(\alpha)$ reads values $\rho(\sigma_{last}(t, \alpha), t)$.*

Consider an execution α and consider a client $c \in \mathcal{C}$, we define as $msg(\alpha, c)$ the set of messages contained in all the client responses in $(\alpha|_{read})|_c$.

The definition below captures the notion that a transaction by a client can only read the initial value ($\perp$) or values that were at some point received by a server.

Definition 4 (Valid values)**.** *Given an execution α and a transaction $t \in comp(\alpha)$, we say that t reads* valid values *if for every $v \in \rho(\sigma_{last}(t, \alpha), t)$, there is an $m \in msg(t, \alpha)$ such that $v \in dec(m)$.*

Well-formed execution. An execution α has *distinct values* if for every two write operations, $write(o, v, d)$ and $write(o', v', d')$ where $o, o' \in \mathcal{O}$ $v, v' \in \mathcal{V}, d, d' \in \mathcal{M}$, it is the case that $v \neq v'$. We can achieve this in practice by having a client append its client identifier and a monotonically increasing counter to the value it intends to write. We say that an execution has *no-transacton reuse* when each client c uses different transactional identifiers for each of c's transactions, as well as different transactional identifiers from other clients.

An execution α is *well-formed* if the following conditions hold for α:

- $\forall p \in \mathcal{S} \cup \mathcal{C}$, $a|_p$ is well-formed;
- α has correct issues, no-transaction reuse, sequential clients, valid responses, and distinct values;
- for every $t \in comp(\alpha)$, transaction t reads valid values;
- if there is a $receive(m, p_j)$ event e' taken by some process p_i in α, then there is an event e that precedes e' in α and $e = send(m, p_i)$ by process p_j;
- for a specific $m \in \mathcal{M}$ if there are z identical events $send(m, p_i)$ taken by process p_j in α, then there are at most z $receive(m, p_j)$ events taken by process p_i in α.

The last two conditions state that a message is not received out of thin air and that there is no message duplication. Both conditions can be implemented in practice with common techniques, such as the use of timestamps [26]. In this paper and unless stated otherwise, we consider only well-formed executions. When we talk about an implementation in our model, we refer to the state machines of all the servers (i.e., function Δ) and all the clients, as well as the sets $\mathcal{S}$, $\mathcal{C}$, $\mathcal{O}$, $\mathcal{V}$, $\mathcal{M}$, and function dec. We denote an *implementation* with $\mathcal{I}$ and say that $\alpha \in \mathcal{I}$ to denote that execution α can be generated by implementation $\mathcal{I}$. Note that if a data store $\mathcal{I}$ can generate an execution α', $\mathcal{I}$ can also generate any execution α where α is a contiguous prefix of α'. Formally, a data store is:

Definition 5 (Data store)**.** *A* data store *is an implementation $\mathcal{I}$ such that for every execution $\alpha \in \mathcal{I}$, α is a well-formed execution.*

Bounded-version data store. In response to a client's read operation to an object o, a server can potentially send an unbounded number of values that were written to o back to the client. In this work, we prove that non-blocking and one round-trip transactions are impossible when a server can only send a bounded number of values to a client. Therefore, we need to define what it means for a data store to be bounded-version. Instead of restricting the number of values a server can send to a client, we allow a server to send an unbounded number of values, and then use (among others) the dec function to restrict the number of values a client can utilize.

Definition 6 (k-version data store)**.** *We say that a data store $\mathcal{I}$ is* k-version *if for every $m \in \mathcal{M}$, $\big|\mathcal{I}.dec(m)\big| \leq k$.*

Although messages are finite, they are unbounded, hence the above definition allows a server to send back an arbitrary long message $m \in \mathcal{M}$ to the client, and hence an unbounded number of values to a client. This allows a client to cache old values and use them in future transactions. However, in combination with valid values (Definition 4) the client can only extract up to a bounded number of values. Note that even if a client uses a cache to store retrieved values, these values cannot be used by the client unless they belong to a decoding of an already received message.

If for a data store $\mathcal{I}$, there is a $k > 0 \in \mathbb{N}$ such that I is a k-version data store, then we say that $\mathcal{I}$ is a *bounded-version data store*, otherwise we say that $\mathcal{I}$ is an *unbounded-version data store*. To the best of our knowledge, function dec is the first one that is able to formally capture the notion of a k-version data store in an elegant way. Other approaches [12], [23], [24] are not formal enough and can be potentially circumvented (see Section V).

Fast reads. In what follows, we define the notion of invisible reads, which refers to the fact that servers do not update their state when they perform a read operation.

Definition 7 (Invisible reads)**.** *A data store $\mathcal{I}$ has* invisible reads, *if for every execution $\alpha \in \mathcal{I}$, for every received $read$ request event or every sent $read$ response event e by some server $s \in \mathcal{S}$, $\sigma = s.\Delta(\sigma, e)$.*

Definition 7 captures the fact that if the state of the server s is σ before event e, then it remains σ after event e takes place. A data store $\mathcal{I}$ that does not provide invisible reads, is said to have *visible reads*. Next, we define non-blocking reads.

We consider function nc that given an execution α and an event $e \in \alpha$ returns a subsequence of α. Formally, consider an execution α, a client c, and client response $e' \in \alpha|_c$, then $nc(\alpha, e')$ corresponds to execution α where all the events between the corresponding request e of e' to a server s and e' are removed from α, except the events that correspond to server s receiving request e and responding e' back to c.

Intuitively, a data store supports non-blocking reads if a server can respond to a read operation without blocking. This means that the server does not have to communicate with other servers in order to respond to the client. Formally:

Definition 8 (Non-blocking reads)**.** *We say that a data store $\mathcal{I}$ has* non-blocking reads, *if for every finite execution $\alpha \in \mathcal{I}$ that ends in a client read response e, then $nc(\alpha, e) \in \mathcal{I}$.*

We now define what it means for a transactional read to take a specific number of rounds. Roughly speaking, an operation takes r rounds, if a clients performs r client responses (i.e., received from the same server) for one client read request.

Definition 9 (r-round reads)**.** *We say that a data store $\mathcal{I}$ has* r-round reads, *if for every execution $\alpha \in \mathcal{I}$, every transaction $t \in \mathcal{T}$, every client c performs at most r client read responses for a specific read request associated with transaction t.*

For instance, in a data store that has 1-round reads this means that a client that issues a transaction t only communicates with a specific server at most once for transaction t.

Definition 10 (Fast reads)**.** *We say that a data store $\mathcal{I}$ has* fast reads *if $\mathcal{I}$ is a* 1*-version data store, and $\mathcal{I}$ has non-blocking and* 1*-round reads.*

Definition 10 captures the notion of latency-optimal reads as informally described by Lu et al. [13], since a client can utilize only one value per read object. We relax the definition of fast reads, by defining what we call semi-fast reads.

Definition 11 (Semi-fast reads)**.** *We say that a data store $\mathcal{I}$ has* semi-fast reads *if $\mathcal{I}$ is a bounded-version data store, and $\mathcal{I}$ has non-blocking and* 1*-round reads.*

In contrast to fast reads, semi-fast reads allow a server to send more than one value back to a client in response to a read request. In this sense, semi-fast reads are not latency-optimal as devised by Lu et al. [13]. In this paper, we prove our impossibility result for semi-fast reads and hence our impossibility result is stronger (i.e., also holds for fast reads).

Monotonic writes. We consider data stores that provide the client-centric consistency model [27] of monotonic writes [28].

Given an execution α and a client $c \in \mathcal{C}$, we define with $ord_w(\alpha, c)$ the set of pairs (e, e') such that e and e' are in $(\alpha|_c)|_{write}$ and e precedes e' in α (and hence in $\alpha|_c$). We define with $ord_w^+(\alpha, c)$ the transitive closure of $ord_w(\alpha, c)$.

Roughly speaking, a data store provides monotonic writes consistency if the write operations performed by a specific client in some specific order, are seen by any other client in this order. Note that monotonic writes do not specify anything regarding the order of write operations between different clients. We use the notation $t \in \alpha$ to denote that there is an event $e \in \alpha$ such that $tx(e) = t$.

Definition 12 (Monotonic writes.)**.** *Consider an execution α and consider every transaction $t \in comp(\alpha)$ with values $\rho(\sigma_{last}(t, \alpha), t) = (v_1, v_2, \ldots, v_r)$. Values $(v_1, v_2, \ldots, v_r)$ are written by client write requests that correspond to events $e_{w_1}, e_{w_2}, \ldots, e_{w_r}$. We say that α* provides monotonic writes *if for any two client write requests e_{w_i} and e_{w_j} with $c = cl(e_{w_i}) = cl(e_{w_j})$ there is no client write request e_w with $cl(e_w) = c$ such that $(e_{w_i}, e_w) \in ord_w^+(\alpha, c)$ and $(e_w, e_{w_j}) \in ord_w^+(\alpha, c)$ and $obj(e_{w_i}) = obj(e_w)$.*

Figure 1 depicts the intuition behind Definition 12. It should not be possible for a transaction reading, among others, objects o and o' to read values v_i and v_j, since the same client wrote value v to object o after writing value v_i to o.

$$\text{client } c: \quad e_{w_i}(o = v_i) \longrightarrow e_w(\xcancel{o = v}) \longrightarrow e_{w_j}(o' = v_j)$$

Fig. 1. A transaction t that reads, among others, objects o and o' cannot read values v_i and v_j due to the existence of e_w. However, transaction t can read values v and v_j for objects o and o' respectively. Note that all three writes e_{w_i}, e_w, and e_{w_j} are performed by the same client c.

Definition 13 (Monotonic writes)**.** *We say that a data store $\mathcal{I}$* provides monotonic writes *if every execution $\alpha \in \mathcal{I}$ has monotonic writes.*

Note that current definitions of monotonic writes [28], [29] refer to single-object operations (i.e., no transactions). Bailis et al. [30] provide an intuitive transactional description but lacks adequate formality. In this sense, this is the first formal definition of monotonic writes in a transactional setting.

Minimal progress. A data store implementation where every read operation performed by a client reads back the initial value of the object $\perp$ is a data store that provides monotonic writes. However, such a data store is of no practical interest. We need to incorporate some notion of liveness in the data store to make it useful. For this, we introduce the notion of minimal progress, that roughly speaking states that if only one client writes a value v to an object o, this value is visible after some some point in time, unless the same client writes a new value or some other client writes o. We start by defining the notion of being eventually responsive. The intuition behind this definition, is that if a client requests a read or write operation, then the client eventually gets a response.

Definition 14 (Eventually Responsive)**.** *A data store $\mathcal{I}$ is* eventually responsive *if for every finite execution $\alpha \in \mathcal{I}$ with last event e that is a client request, every infinite extension $\alpha' \in \mathcal{I}$ of α has a client response e' such that e''s corresponding request is e.*

Another way to think of the above definition is that given a finite execution α that ends with a request to a client, there is an extension of α that contains the response to the client.

We say that a *transaction t appears for the first time* after an event e' in an execution α if there is no event $e \in \alpha$ that appears before e' in α with $tx(e) = t$. The following definition captures the idea that from some point onwards, all transactions keep returning the newer written values. If a single write with value v is performed to an object o, then there is a point after which all transactions that appear for the first time read v or a later written value for object o.

Definition 15 (Eventually Visible)**.** *A data store $\mathcal{I}$ is called* eventually visible *if for every finite execution $\alpha \in \mathcal{I}$ with last event e_w that is a client write request to object o. Consider all $r > 0$ completed client write requests before e_w to o: $e_{w_1}, \ldots, e_{w_r}$ in α. In other words, $cor(e_{w_1}), \ldots, cor(e_{w_r})$ appear before e_w in α. Each of these client write requests, writes a value. Consider these values to be contained in the set v_{old}. Then, there is a bound $b > 0$, such that for every extension $\alpha' \in \mathcal{I}$ of α, every transaction that appears for the first time after $|\alpha| + b$ in α' and that requests o, does not read a value that belongs to $v_{old} \cup \{\perp\}$ for o.*

Definition 16 (Minimal Progress)**.** *A data store $\mathcal{I}$ provides* minimal progress *if $\mathcal{I}$ is eventually responsive and visible.*

Note that if a data store $\mathcal{I}$ provides minimal progress, $\mathcal{I}$ cannot use the stable snapshot approach [22]. In the stable snapshot approach, servers totally order write operations, as well as servers keep track of the most recent stable snapshot. A stable snapshot is a point in the serial order of the write operations, for which all updates are known to have been applied to all objects. When reading, a client c indicates the last known stable snapshot, and then c reads from that snapshot and retrieves information about the current last known stable snapshot (that c uses in the next transaction). Thus client c can always make progress, albeit by reading from the past. However, using the stable snapshot approach, when a client c that has been inactive (i.e., not performing any operations) for an arbitrary long amount of time, issues a new transaction, c could potentially read old values and violate eventual visibility.

III. Fast Transactions Are Impossible

In this section, we prove that fast transactions are impossible. Specifically, we prove our main result.

Theorem 1. *No data store can provide monotonic writes, minimal progress, and invisible semi-fast reads.*

Fast reads. For pedagogical purposes, we first present the proof of the following theorem:

Theorem 2. *No data store can provide monotonic writes, minimal progress, and invisible fast reads.*

For our result, we consider two servers $s_1, s_2 \in \mathcal{S}$ and two objects $o_1, o_2 \in \mathcal{O}$ served by servers s_1 and s_2 respectively. Furthermore, we consider three clients $c_r, c_h, c_w \in \mathcal{C}$, where client c_h issues a finite number of transactions where each transaction reads both objects o_1 and o_2, client c_r issues a single transaction t to both objects o_1 and o_2, and client c_w performs writes. Before we continue, we introduce some auxiliary notation to simplify the proof.

A read operation in a data store with fast reads consists of 4 events e_1, e_2, e_3, e_4 in this order. Event e_1 is $send(read(o, t_i, m_s), s)$, event e_2 is $receive(read(o, t_i, m_s), c)$, and since a data store with fast reads is non-blocking, this means the server s can respond right away to the client with $e_3 = send(res((o, m_r)), c)$, and finally the client c receives the message $e_4 = receive(res((o, m_r)), s)$. In what follows we are going to denote event e_1 as req_o and the remaining three events as $resp_o(m_r)$, meaning that for object o the client got back response m_r.

Similarly, we denote with $w_o(v)$ the sequence of events needed to write value v to object o. In contrast to a read operation event, a write operation event might span an arbitrary, however finite, number of events. Arbitrary since write operations are not necessarily non-blocking and hence a server might communicate with other servers before completing the write. Finite since we consider a data store with eventual responsiveness (i.e., the write should eventually complete).

For the proof of Theorem 2 we assume by way of contradiction that a data store exists that provides monotonic writes, minimal progress, and invisible fast reads. For this, we consider execution

$$\alpha = w_{o_1}(v^1_{o_1}), w_{o_2}(v^1_{o_2}), t_1, \ldots, t_{l_1}, \\ w_{o_1}(v^2_{o_1}), w_{o_2}(v^2_{o_2}), t_{l_1+1}, \ldots, t_{l_2}$$

where the first two writes are performed by client c_w and transactions $t_1, \ldots, t_{l_1}$ are performed by client c_h until values $v^1_{o_1}$ and $v^1_{o_2}$ become visible. Then, client c_w performs two more writes to objects o_1 and o_2 and afterwards client c_h takes steps until values $v^2_{o_1}$ and $v^2_{o_2}$ are visible.

Based on execution α, we construct executions α_1 and α_2:

$$\alpha_1 = w_{o_1}(v^1_{o_1}), w_{o_2}(v^1_{o_2}), t_1, \ldots, t_{l_1}, w_{o_1}(v^2_{o_1}), w_{o_2}(v^2_{o_2}), \\ t_{l_1+1}, \ldots, t_{l_2}, req_{o_1}, req_{o_2}, resp_{o_1}(m^1_1), resp_{o_2}(m^1_2)$$

$$\alpha_2 = w_{o_1}(v^1_{o_1}), w_{o_2}(v^1_{o_2}), t_1, \ldots, t_{l_1}, req_{o_1}, req_{o_2}, \\ resp_{o_1}(m^2_1), w_{o_1}(v^2_{o_1}), w_{o_2}(v^2_{o_2}), t_{l_1+1}, \ldots, t_{l_2}, resp_{o_2}(m^2_2)$$

Note the differences between executions α_1 and α_2 (see Figure 2). In execution α_1, client c_r performs both read operations on objects o_1 and o_2 when the values $v^2_{o_1}$ and $v^2_{o_2}$ are visible. In execution α_2, c_r read requests are sent after values $v^1_{o_1}$ and $v^1_{o_2}$ are visible. However, the request to object o_1 is received by s_1 before value $v^2_{o_2}$ is visible and the request to object o_2 is received by s_2 after value $v^2_{o_2}$ is visible.

In execution α_1 client c_r receives only two messages, m^1_1 and m^1_2. Message m^1_1 cannot contain a value for object o_2 since o_2 is not served by server s_1 and s_1 can only contain values for object o_1. Therefore, for c_r to read value $v^2_{o_2}$ for object o_2, it should be the case that message m^1_2 in execution α_1 contains $v^2_{o_2}$. Assume by way of contradiction that m^1_2 does not contain $v^2_{o_2}$, this means that there is no way for client c_r to have read value $v^2_{o_2}$, therefore $v^2_{o_2} \notin \rho(\sigma_{last}(t, \alpha_1), t)$, hence the value is not visible yet. A contradiction, therefore $v^2_{o_2} \in dec(m^1_2)$.

In execution α_2 note that m^2_1 cannot contain $v^2_{o_1}$ (due to the written-values responses property) since at this point in execution α_2, $v^2_{o_1}$ has not been written yet. We argue that message m^2_2 in execution α_2 should contain value $v^1_{o_2}$. Assume it does not. Then, the only other possible values m^2_2 can contain are $\perp$ and $v^2_{o_2}$ (recall that we consider fast reads, hence 1-version reads). However the pair $(v^1_{o_1}, \perp)$ would not satisfy minimal progress since value $v^1_{o_2}$ is visible when client c_r performed its transaction. Furthermore, pair $(v^1_{o_1}, v^2_{o_2})$ violates monotonic writes, since the same client wrote $v^2_{o_1}$ to object o_1 before writing $v^2_{o_2}$ to object o_2. A contradiction in both cases. Therefore $v^1_{o_2} \in dec(m^2_2)$.

Executions α_1 and α_2 are indistinguishable to server s_2. Indeed, in both executions α_1 and α_2, server s_2 receives, performs, and responds to the client's c_r request ($resp_{o_2}$) to read object o_2 after values $v^2_{o_1}$ and $v^2_{o_2}$ have been written and are visible. Since all the transactions performed by client c_h are invisible, server s_2 cannot distinguish between the executions and respond back to client c_r in the same way in both executions, and therefore messages m^2_2, and m^1_2 are equal ($m^1_2 = m^2_2 = m_2$). Since we consider distinct values, $v^1_{o_2} \neq v^2_{o_2}$, and both are in $dec(m_2)$, it is the case that $\left|dec(m_2)\right| = 2 > 1$, a contradiction. Hence, Theorem 2 holds.

Semi-fast reads. We prove Theorem 1 by contradiction. We assume that a data store $\mathcal{I}$ exists that provides monotonic writes, minimal progress, and invisible semi-fast reads. Specifically, we assume that data store $\mathcal{I}$ is a k-version data store. We prove that there is an execution where server s_2 sends a message back to a client that contains more than k values in order for $\mathcal{I}$ to satisfy monotonic writes. To prove our impossibility result, we construct $k+1$ executions $\alpha_1, \alpha_2, \ldots, \alpha_{k+1}$ and show that all these executions are indistinguishable to server s_2. We show that in execution α_i, server s_2 needs to respond with a message that contains value $v^i_{o_2}$.

Before presenting the construction of executions α_i ($1 \leq i \leq k+1$), we first construct execution $\alpha \in \mathcal{I}$ in which all executions α_i are based upon. We construct execution α as

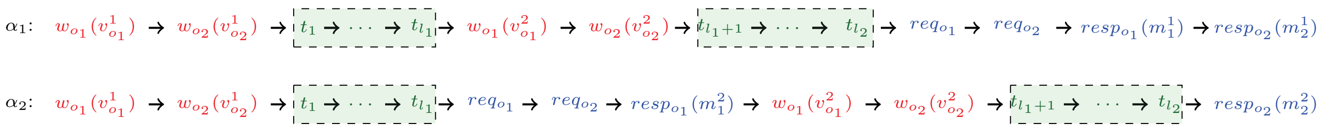
α_1: $w_{o_1}(v_{o_1}^1) \rightarrow w_{o_2}(v_{o_2}^1) \rightarrow t_1 \rightarrow \cdots \rightarrow t_{l_1} \rightarrow w_{o_1}(v_{o_1}^2) \rightarrow w_{o_2}(v_{o_2}^2) \rightarrow t_{l_1+1} \rightarrow \cdots \rightarrow t_{l_2} \rightarrow req_{o_1} \rightarrow req_{o_2} \rightarrow resp_{o_1}(m_1^1) \rightarrow resp_{o_2}(m_2^1)$
α_2: $w_{o_1}(v_{o_1}^1) \rightarrow w_{o_2}(v_{o_2}^1) \rightarrow t_1 \rightarrow \cdots \rightarrow t_{l_1} \rightarrow req_{o_1} \rightarrow req_{o_2} \rightarrow resp_{o_1}(m_1^2) \rightarrow w_{o_1}(v_{o_1}^2) \rightarrow w_{o_2}(v_{o_2}^2) \rightarrow t_{l_1+1} \rightarrow \cdots \rightarrow t_{l_2} \rightarrow resp_{o_2}(m_2^2)$

Fig. 2. Executions α_1 and α_2 where client c_w writes (in red) objects o_1 and o_2. Client c_h issues a finite number of transactions (in green) between the c_w's writes and c_r performs a transaction that reads both objects o_1 and o_2 (in blue).

follows. First, client c_w writes value $v_{o_1}^1$ to object o_1, waits for the response of server s_1 to acknowledge the write, and then writes value $v_{o_2}^1$ to object o_2 and waits for server s_2 to acknowledge the write. Servers s_1 and s_2 acknowledge the writes since $\mathcal{I}$ provides minimal progress, and hence $\mathcal{I}$ is eventually responsive. Afterwards, client c_h issues transactions until values $(v_{o_1}^1, v_{o_2}^1)$ are visible, again due to minimal progress. After the values are visible, we allow c_w to write value $v_{o_1}^2$ to object o_1 and afterwards value $v_{o_2}^2$ to object o_2. Then, we allow client c_h to issue transactions until values $(v_{o_1}^2, v_{o_2}^2)$ are visible. We keep repeating the same procedure until values $(v_{o_1}^{k+2}, v_{o_2}^{k+2})$ are visible. Namely c_w writes both objects o_1 and o_2 and then c_h issues transactions until the latest written values by c_w become visible. Execution α (see top of Figure 3) is therefore:

$$\alpha = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), t_1, \ldots, t_{l_1}, w_{o_1}(v_{o_1}^2), w_{o_2}(v_{o_2}^2),$$
$$t_{l_1+1}, \ldots, t_{l_2}, w_{o_1}(v_{o_1}^3), w_{o_2}(v_{o_2}^3), \ldots,$$
$$w_{o_1}(v_{o_1}^{k+2}), w_{o_2}(v_{o_2}^{k+2}), t_{l_{k+1}+1}, \ldots, t_{l_{k+2}}$$

Before we continue with each individual execution α_i, we prove the following lemma for transactions in execution α, where we consider that $v_{o_1}^0 = v_{o_2}^0 = \perp$.

Lemma 1. *No transaction introduced in execution α that appears for the first time after transaction t_{l_i} ($i > 0$) completes can read values $v_{o_1}^j$ and $v_{o_2}^j$ with $0 \le j < i$ for objects o_1 and o_2 respectively.*

Proof: By construction of execution α, we know that after transaction t_{l_i}, values $v_{o_1}^i$ and $v_{o_2}^i$ are visible. By the definition of minimal progress, no transaction that appears for the first time after transaction t_{l_i} has completed can return an older value for objects o_1 and o_2. Hence no transaction after t_{l_i} can read values $v_{o_1}^j$ and $v_{o_2}^j$ with $0 \le j < i$. ■

Lemma 2. *No transaction introduced in α that reads objects o_1 and o_2 can read values $(v_{o_1}^a, v_{o_2}^b)$ with $a < b$.*

Proof: Assume by way of contradiction that we can introduce a transaction t in α that reads $(v_{o_1}^a, v_{o_2}^b)$ with $a < b$. If $a < b$, then since client c_w issues writes in this order $v_{o_1}^a \rightarrow \cdots \rightarrow v_{o_1}^b \rightarrow v_{o_2}^b$, transaction t that reads $(v_{o_1}^a, v_{o_2}^b)$ with $a < b$ violates monotonic writes, a contradiction, since t should have read value $v_{o_1}^b$ for o_1. ■

We construct execution α_i for $1 \le i \le k+1$ as follows (with $l_0 = 0$):

$$\alpha_i = w_{o_1}(v_{o_1}^1), w_{o_2}(v_{o_2}^1), \ldots, w_{o_1}(v_{o_1}^i), w_{o_2}(v_{o_2}^i),$$
$$t_{l_{i-1}+1}, \ldots, t_{l_i}, req_{o_1}, req_{o_2}, resp_{o_1}(m_1^i),$$
$$w_{o_1}(v_{o_1}^{i+1}), w_{o_2}(v_{o_2}^{i+1}), \ldots, w_{o_1}(v_{o_1}^{k+2}), w_{o_2}(v_{o_2}^{k+2}),$$
$$t_{l_{k+1}+1}, \ldots, t_{l_{k+2}}, resp_{o_2}(m_2^i)$$

Lemma 3. *Value $v_{o_2}^i \in dec(m_2^i)$ in execution α_i.*

Proof: Assume by way of contradiction that $v_{o_2}^i \notin dec(m_2^i)$ in execution α_i. Due to Lemma 1, transaction t can only read values $(v_{o_1}^a, v_{o_2}^b)$ with $a \ge i$. Since the read to o_1 is performed before the write of value $v_{o_1}^{i+1}$ takes place, this means that t should read value $v_{o_1}^i$ for object o_1. Therefore the only allowed values for object o_2 is $v_{o_2}^i$ since returning $v_{o_2}^j$ with $j < i$ implies that $v_{o_2}^i$ is not visible yet (Lemma 1) and returning $v_{o_2}^j$ with $j > i$ violates monotonic writes (Lemma 2). In other words, if $v_{o_2}^i \notin dec(m_2^i)$, transaction t has no correct values to return. A contradiction. ■

Due to Lemma 3, we know that $v_{o_2}^i \in dec(m_2^i)$ for every i, $1 \le i \le k+1$. However, all executions α_i are indistinguishable to server s_2 and therefore $m_2 = m_2^i$ for every $1 \le i \le k+1$. Due to the distinct values property, all values $v_{o_2}^i \neq v_{o_2}^j$ for $i \neq j$ and hence $v_{o_2}^i \in dec(m_2)$ implies that $\left|dec(m_2)\right| > k$, a contradiction.

Circumventing the impossibility. In a synchronous system, where the duration of a transaction cannot span more than one synchronous round, Theorem 1 collapses. Meaning that in a synchronous system, we can have a data store with fast invisible reads that also provides minimal progress and monotonic writes. In such a scenario, every server stores the latest value written to an object. In each round, in case of a read, the server returns this latest value, and in case of a write it overwrites the currently stored value. However, highly available data stores [2], [3], [7] are deployed in the wide-area (i.e., not synchronous setting), since synchronous settings are not realistic, because among others, they prevent the possibility of partitions. Additionally, in a synchronous system we need to choose the duration of rounds in a conservative manner which contradicts the idea of having transactions as fast as possible.

Fast transactions and fault-tolerance. At first sight, the ramifications of Theorem 1 are inconspicuous. Someone might think that whether transactions are visible or not has little impact on the latency of transactions. This might be the case if we assume that the client communicates with a distant server. Nevertheless, in practice, we have to consider the possibility that if a visible transaction updates the state of a server s, s might fail (i.e., crash). In case server s fails, we might lose all the information that was written by s during the update. Highly available data stores have to implement fail-over, therefore, in

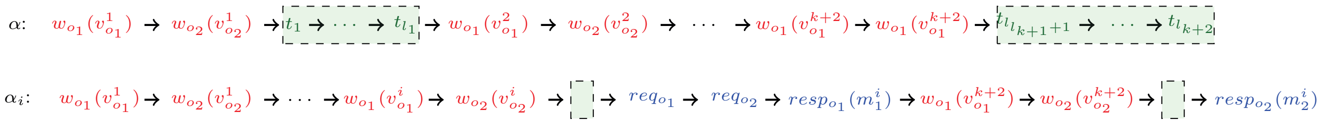

Fig. 3. At the top, we depict execution α where client c_w alternates between writing (in red) objects o_1 and o_2. Client c_h issues a finite number of transactions (in green) after client's c_w write of value $v^j_{o_2}$ until values $(v^j_{o_1}, v^j_{o_2})$ are visible. At the bottom, we depict execution α_i. Due to space constraints, we depict the transactions of c_h until values $(v^i_{o_1}, v^i_{o_2})$ and $(v^{k+2}_{o_1}, v^{k+2}_{o_2})$ are visible with shortened green boxes. The events of client's c_r transaction that read objects o_1 and o_2 are depicted in blue.

case server s fails, another server s' should take over in place of s. However, server s' does not contain the data that was written during the update. The only way for server s' to know about the updated state of s, is if s replicated the update to s' before failing. In such a scenario, the transaction is blocking since it has to wait for the underlying writes, to be replicated before responding to the client. To summarize, a data store with fast transactions cannot be fault-tolerant, and conversely, a fault-tolerant data store cannot provide fast transactions.

IV. Unbounded-Version Data Store

Theorem 1 states that we cannot devise a bounded-version data store with non-blocking and 1-round reads. This raises the question on whether we can achieve non-blocking and 1-round reads in an unbounded-version data store. With Theorem 3 we answer this question in the affirmative.

Theorem 3. *There is an unbounded-version data store that provides monotonic writes, minimal progress, non-blocking and 1-round reads.*

To prove Theorem 3, we devise `ubvStore`, a new unbounded-version data-store with non-blocking and 1-round reads. `ubvStore` is unbounded-version since a server can send an unbounded number of values to a client (i.e,. $\nexists k : \forall m \in \mathcal{M}\ \big|dec(m)\big| < k$). Note that we do not claim that `ubvStore` is a practical algorithm. We use `ubvStore` to generalize our impossibility result (Theorem 1).

Overview. We base `ubvStore` on three ideas: (i) servers assign version numbers to values, (ii) servers and clients store locally the write history (i.e., sequence of writes performed by a client) of each client and exchange write histories whenever they communicate, and (iii) a read-only transaction by a client c is performed on c's local knowledge of the write histories. We describe in detail these three ideas below.

An object o is served by a single server s ($o \in s.obj$). Therefore, server s can *order* the writes issued by clients to an object o by assigning incrementally increasing *version numbers* to values written to an object o. We say that a value v written to an object o, has version number vn, if v was the vn-th value written to object o. For example, if two writes to object o with values v_1 and v_2 are performed by two different clients c_{w_1} and c_{w_2} respectively, s can assign version number 1 to value v_1 and version number 2 to value v_2. The initial value $\perp$ of each object has version number 0.

Both servers and clients in `ubvStore` contain local information about the write history of each client. The write history of a client c corresponds to the ordered list of the write operations client c has performed. Specifically, the write history of a client is a list of triples, where each triple is of the format $(object, value, version\ number)$. We denote with $hist_p[c]$ the write history of client c as it is known to process p. Naturally, note that process p might have stale information on the write history of client c and hence $hist_p[c]$ is a subset of client's c actual write history. For example, in a system with one server s and two clients c_1 and c_2, server s might store locally $hist_s[c_1] = (o_1, v_1, 3) \cdot (o_2, v_3, 1)$ and $hist_s[c_2] = \epsilon$. This means that s is aware that client c_1 first performed the write of value v_1 to object o_1 and then the write of value v_3 to object o_2, and since $hist_s[c_2] = \epsilon$, this means that s has no information about the write history of client c_2 (potentially client c_2 has not performed any writes). Similarly, client c_1 might locally contain $hist_{c_1}[c_2] = (o_5, v_9, 6)$.

Whenever a client c performs a write operation to an object served by a server s, c sends all its write histories ($hist_c$) to server s. Similarly, when a server s responds to the read request of a client c, s sends all its write histories ($hist_s$) to client c. This way, whenever servers and clients communicate, they can potentially *extend* their local write histories. Note that a server stores only values (i.e., versions) of objects that it serves. Nevertheless, a server s can store the write histories of all the clients, and which objects these clients wrote, even though s might not serve these objects. For example, a server s with $o_1 \notin s.obj$ could contain $hist_s[c_3] = (o_1, \_, 9)$ where _ implies that s does not "know," and hence does not store the value of object o_1 since s does not serve o_1.

In `ubvStore`, transactions are 1-round and non-blocking. This means that when a client c performs a transaction, c sends messages to the servers serving the transaction's objects and the servers respond immediately to c. Then, client c retrieves the responses from the servers and potentially extends $hist_c$. Afterwards, c attempts to read the latest (i.e., by looking at the version number) values of each object it wants to read without violating monotonic writes. If it can read the latest values without violating monotonic writes, then client c reads these latest values. If not, client c attempts to read potentially earlier values of objects until it finds a tuple of values that satisfies monotonic writes.

To give an example of how `ubvStore` performs a transaction, consider execution α_2 in Figure 2. In execution α_2, client c_r sends messages to both s_1 and s_2 as part of transaction t. Server s_1 receives the message after value $v^1_{o_1}$ is visible and before value $v^2_{o_1}$ is written. Therefore, s_1 sends value $v^1_{o_1}$ to c_r. Server s_2 receives c_r's message after value $v^2_{o_2}$ is visible and hence s_2 sends value $v^2_{o_2}$ to c_r. The issue with execution α_2 is that values $(v^1_{o_1}, v^2_{o_2})$ cannot be read for t since these values violate monotonic writes. However, since we consider an unbounded-version data store, server s_2 can also send value $v^1_{o_2}$ to c_r. In `ubvStore`, when client c_r receives the message from server s_1, c_r has $hist_{c_r}[c_w] = (o_1, v^1_{o_1}, 1)$. When c_r receives the message from s_2, c_r knows that $hist_{c_r}[c_w] = (o_1, v^1_{o_1}, 1) \cdot (o_2, v^1_{o_2}, 1) \cdot (o_1, \_, 2) \cdot (o_1, v^2_{o_1}, 2)$. Knowing $hist_{c_r}$, client c_r can safely read values $(v^1_{o_1}, v^1_{o_2})$ for transaction t.

`ubvStore` in detail. The algorithm of `ubvStore` is presented in algorithms 1 and 2. Lines 1-48 correspond to the code for the client and lines 49-60 correspond to the server code. `ubvStore` uses both procedures, as well as event-based (i.e., **trigger** and **upon event**) techniques [26]. A trigger event corresponds to the transmission of a message, while an upon event corresponds to the receipt of a message. Note that algorithms 1 and 2 do not contain the conceptually simple EXTEND, MAXVERNUMBER, GETVALUES, and CLEAN functions due to space constraints, but we explain how they operate. The code of these functions appears in the technical report [25]. A client cl has the following local variables (lines 2-4): $hist_{cl}$, $responses$, and $versionNumber$. Client cl stores the write histories in $hist_{cl}$, that is an array of lists, where each list (except $hist_{cl}[c_{init}]$ – see below) is initially empty (ϵ). For a list $list$, we denote with $|list|$ its length. We consider that there is some initial client $c_{init} \notin \mathcal{C}$ that wrote value $\perp$ with version number 0 to all the objects. This way, if a client c performs a transaction on an object o that has not been written, the initial value $\perp$ is read.

Variables $responses$ and $versionNumber$ are used in the READ and WRITE procedures in order to inform the client that a message has been received from the server. Specifically, $responses$ contains the messages received from servers during a read transaction. $versionNumber$ contains the version number the server assigns to the object the client writes.

Client cl performs a transaction by calling the READ procedure (lines 6-23) and providing as parameters the objects O_s, a transactional identifier t, and an additional message d. For every server s that serves an object $o \in O_s$, client cl triggers a server READ event (lines 7-9). Note that client cl only "asks" from a server s the objects that s serves using the OBJECTSSERVEDBY procedure. We assume that OBJECTSERVEDBY returns the set of all the objects served by s and this information is stored locally in cl (i.e., no communication with the server is needed). The client then waits until it receives messages from all the servers it sent a message to (Line 10). For this, client cl uses the $responses$ variable that is initially ϵ and each time cl receives a response (READRESPONSE) from the server, the response is appended to $responses$ (lines 34-35). When cl receives messages from all the servers it has communicated with, then $|responses|$ is equal to $|servers|$ (Line 10). After receiving the responses from the servers, client cl calls the EXTEND procedure

Algorithm 1 `ubvStore` for a client cl

```
1:  ▷ Client's $cl$ local variables
2:  $hist_{cl}[c] \leftarrow \epsilon \ \forall c \in \mathcal{C}$
3:  $hist_{cl}[c_{init}] \leftarrow (o_1, \perp, 0) \cdot (o_2, \perp, 0) \cdot \dots \cdot (o_k, \perp, 0)$
4:  $responses \leftarrow \epsilon$, $versionNumber \leftarrow -1$
5:
6:  procedure READ($O_s, t, d$)  ▷ $cl$ to read objects in $O_s$
7:      for server $s$ in {server that serves an object $o \in O_s$} do
8:          ▷ ask server $s$ only of objects in $O_s$ that $s$ serves
9:          trigger $\langle s, \text{READ} \mid O_s \cap \text{OBJECTSSERVED}(s), t, d \rangle$
10:     wait until $|responses| = |servers|$
11:     $hist_{cl} \leftarrow \text{EXTEND}(hist_{cl}, responses)$
12:
13:     $verToRead[o] \leftarrow 0 \ \forall o \in \mathcal{O}$
14:     for object $o$ in $O_s$ do
15:         $verToRead[o] \leftarrow \text{MAXVERNUMBER}(hist_{cl}, o)$
16:     while true do
17:         $result \leftarrow \text{GETVALUES}(hist_{cl}, verToRead)$
18:         $(safe, prObj) \leftarrow \text{ISSAFE}(hist_{cl}, result)$
19:         if $safe$ then
20:             return $result$
21:         else
22:             $verToRead[prObj] \leftarrow verToRead[prObj] - 1$
23:     $responses \leftarrow \epsilon$
24:
25: procedure WRITE($o, v, d$)  ▷ $cl$ to write value $v$ to object $o$
26:     $s \leftarrow$ server that serves $o$
27:     $histToSend \leftarrow \text{CLEAN}(hist_{cl}, s)$
28:     trigger $< s, \text{WRITE} \mid o, v, histToSend >$
29:     wait until $versionNumber \neq -1$
30:     $hist_{cl}[c] \leftarrow hist_{cl}[c] \cdot (o, v, versionNumber)$
31:     $versionNumber \leftarrow -1$
32:     return $ack$
33:
34: upon event $< cl, \text{READRESPONSE} \mid r >$ do
35:     $responses \leftarrow responses \cdot r$
36:
37: upon event $< cl, \text{WRITERESPONSE} \mid vn >$ do
38:     $versionNumber \leftarrow vn$
39:
40: procedure ISSAFE($hist_{cl}, result$)
41:     if $\exists o : o.val = \_ \wedge o \in result$ then
42:         return (false, $o$)
43:     for client $c$ in $\mathcal{C}$ do
44:         ▷ triple $tr \leftarrow (val, o, vn)$, $v(tr) = val \wedge obj(tr) = o$
45:         if $\exists r_1, r_2, r_3 \in hist_{cl}[c] : v(r_1), v(r_3) \in result$ then
46:             if $r_1 \rightarrow r_2 \wedge r_2 \rightarrow r_3 \wedge obj(r_1) = obj(r_2)$ then
47:                 return (false, $obj(r_3)$)
48:     return true
```

Algorithm 2 `ubvStore` for a server s

```
49: ▷ Server's $s$ local variables
50: $hists_s[c] \leftarrow \epsilon \ \forall c \in \mathcal{C}$
51: $mem[o] \leftarrow (\perp, 0) \ \forall o$ served by $s$
52:
53: upon event $< s, \text{READ} \mid O_s, t, d >$ do
54:     trigger $< cl, \text{READRESPONSE} \mid hist_s >$
55:
56: upon event $< s, \text{WRITE} \mid o, v, hist_{cl} >$ do
57:     $mem[o] \leftarrow (v, mem[o].ver + 1)$
58:     $hists_s[c] \leftarrow hists_s[c] \cdot (o, v, mem[o].ver)$
59:     $hists_s \leftarrow \text{EXTEND}(hists_s, hist_{cl})$
60:     trigger $< cl, \text{WRITERESPONSE} \mid mem[o].ver >$
```

(Line 11) that extends $hist_{cl}$ if cl received potentially additional information from some of the servers (e.g., received information such that cl can extend a write history $hist_{cl}[c]$ for some client c). The EXTEND procedure takes two write histories and combines them to get the maximum possible write histories. Afterwards, client cl stores to the $verToRead$ array all the versions that it intends to read from this transaction. Initially, cl intends to get the latest version of each object (lines 14-15) and cl is able to retrieve the latest version number (as known by cl) of each object using the MAXVERNUMBER procedure. The MAXVERNUMBER procedure accepts write histories ($hist$) and an object o and finds the maximum version number of object o by utilizing the information in $hist$.

Client cl then enters the **loop** (lines 16-22), where cl attempts to read the latest values that satisfy monotonic writes, until it succeeds. It does so by calling the GETVALUES procedure. The GETVALUES procedure loops through all the local write histories to find the value of object o with the specific version number. Then, the client verifies that the read values are safe (i.e., satisfy monotonic writes) by calling ISSAFE (Line 18). The ISSAFE procedure (lines 40-48) takes as input the write histories ($clientHist$) and the read values ($result$) and examines whether the read values by cl violate monotonic writes (Line 46). Note that in ISSAFE client cl might get back _ as the value of an object (Line 41), meaning that cl knows that a value was written by some client but is not aware of this value. In such a case, client cl attempts to read the immediately previous value of that object in the next loop, by decrementing the version that is about to be read for that object (Line 22) and repeating the loop. At the end, the client sets $responses$ back to ϵ (Line 23).

Specifically, ISSAFE checks whether two values $v(t_1)$ and $v(t_3)$ have been read by the client, and value $v(t_1)$ of $obj(t_1)$ has been overwritten before the write of value $v(t_3)$. If this is the case, the values violate monotonic writes and ISSAFE returns $obj(t_3)$ (Line 47). Note that monotonic writes are violated because the client attempted to read the version of $obj(t_3)$, and therefore we call $obj(t_3)$ the problematic object. If the read values are safe (Line 48), the read values are returned (Line 20). Otherwise, the version needed for object $prObj$ is decremented by one (Line 22) and the loop repeats.

To perform a write operation, client cl calls the WRITE procedure (lines 25-32). Client cl retrieves the server that serves object o (Line 26), cleans history $hist_{cl}$ by removing values of objects server s does not serve (Line 27) and then cl sends a WRITE message to the server (Line 28). Recall that a server does not store values of objects that it does not serve. Therefore the client calls the CLEAN procedure that simply goes through all the triples in the provided history and if the history of some client contains a value that is not served by the server it stores _ as the value. Afterwards, the client waits until the server responds (Line 29) (i.e., $versionNumber$ is updated when receiving a WRITERESPONSE from the server (lines 37-38). Finally, client cl updates its write histories from the one received by the client (Line 30), sets $versionNumber$ back to -1 and returns an acknowledgment.

The code for a server s is presented in Algorithm 2. As with a client, server s stores locally all the write histories ($hist_s$). Additionally, s stores the values of the objects it serves in the mem array (Line 51). For each object $o \in s.obj$, $mem[o]$ corresponds to a pair with the latest written value and its corresponding version number. When s receives a READ event, it responds to the client with a READRESPONSE that contains $hist_s$ (Line 54). Note that s does not update its state in response to a $read$ event, and hence reads in `ubvStore` are invisible. When s receives a WRITE event to write a value v to object o (Line 56), s stores the value in $mem[o]$ and increases its version number by one (Line 57). Then s (potentially) extends its current knowledge of $hist_s$ by the one received ($hist_{cl}$) from the client using the EXTEND procedure (Line 59). Finally, server s responds to the client with a WRITERESPONSE including a version number (Line 60).

The cautious reader might notice that some parameters (e.g., d) are not used. Nevertheless, such parameters appear in `ubvStore` to adhere to the model we present in Section II.

Correctness. To prove that `ubvStore` is correct, we have to show that `ubvStore`: (i) is a valid data store (i.e., all its executions are well-formed), (ii) provides monotonic writes, and (iii) provides minimal progress. We refer the reader to the extended version of the paper [25] for the proofs.

As a final remark, note that in practice, sending several versions in one round is *costly*. Hence, we expect that a `ubvStore` implementation would perform worse than COPS-SNOW [13]. However, this is not an apples to apples comparison, because COPS-SNOW can violate consistency during asynchrony, while `ubvStore` never violates consistency.

V. Concluding Remarks

Our framework is inspired by the models of Attiya et al. [10] and Burckhardt et al. [31]. However, in addition, our framework captures transactions. As far as we know, our formal framework is the first to precisely capture the notion of fast transactions, and specifically the notion of bounded-version data stores. We formally defined bounded-version data stores by introducing the decoding function dec and the definition of valid values, something we believe is a contribution in itself. In contrast, other models [12], [13], [23] that attempt to define fast transactions, fail to precisely define the restrictions imposed by one-version reads (i.e., servers could potentially "cheat" and respond with more than one version).

Lu et al. [13] introduce the informal notion of latency-optimal read-only transactions (i.e., fast reads in Definition 10), as well as present the SNOW theorem. The SNOW theorem states that we cannot devise a read-only transaction algorithm that satisfies the following four properties: (i) strict serializability, (ii) non-blocking reads, (iii) one-round and one-version reads, and (iv) coexistence with write transactions. Konwar et al. [23] revisit the SNOW theorem and present some new results that consider the role of client-to-client messaging to the SNOW theorem. Tomsic et al. [22] investigate the relation between the consistency, the speed, and the freshness of reads and among others show that visible fast transactions are possible by reading an arbitrarily old snapshot of the database and hence such a data store does not provide the weak property of eventual visibility. Didona et al. [12], [24] look into causally-consistent data stores and investigate the cost fast read-only transactions impose on writes, as well as prove that a data store cannot provide fast read-only transactions in combination with write transactions. In contrast to previous work [12], [13], [22]–[24], we consider the weaker consistency model of monotonic writes and hence we strengthen our impossibility result. Finally, note that real systems [32]–[34] that provide fast transactions, either assume strong synchrony (e.g., Spanner [34] relies on global time), or provide visible read-only transactions and hence are not fault-tolerant.

Finally, to the best of our knowledge, our work is the first to bring fault-tolerance and fast transactions together into attention. As a matter of fact, the original work of Lu et al. [13] that informally introduces fast transactions, presents the COPS-SNOW data store that supports such fast transactions. COPS-SNOW is able to keep operating during a network partition and hence the loss of one or more data centers. However, COPS-SNOW cannot tolerate the failure of even a single server, since otherwise consistency is violated.

In this paper, we proved that invisible fast transactions are impossible in a data store that supports the weak consistency model of monotonic writes. To prove our result, we devised a formalization to precisely capture notions such as one-round, one-version, etc. Our proof is the first one to shed light on an important and unexplored consequence of fast transactions, that is, a data store that supports fast transactions cannot tolerate the failure of even one server. Additionally, with `ubvStore`, we showed that the number of versions a server can send to a client is consequential to whether transactions can be both non-blocking and 1-round trip.

References

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[2] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[3] MongoDB. https://www.mongodb.com. [Online; accessed 14-October-2019].

[4] Eric A. Brewer. Towards robust distributed systems (abstract). In *PODC*, 2000.

[5] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[6] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20:20–20:32, July 2014.

[7] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *CUFP*, 2010.

[8] Paul R Johnson and Robert Thomas. Maintenance of duplicate databases. RFC 677, 1975.

[9] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report TR-11-21, The University of Texas at Austin, 2011.

[10] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):141–155, 2017.

[11] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.

[12] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Causal consistency and latency optimality: friend or foe? In *VLDB*, 2018.

[13] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *OSDI*, 2016.

[14] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *SOSP*, 2011.

[15] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Optimistic causal consistency for geo-replicated key-value stores. In *ICDCS*, 2017.

[16] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *DSN*, 2018.

[17] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. Paris: Causally consistent transactions with non-blocking reads and partial replication. In *ICDCS*, 2019.

[18] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *NSDI*, 2017.

[19] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SoCC*, 2014.

[20] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, 2016.

[21] 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. TAO: Facebook's distributed data store for the social graph. In *ATC*, 2013.

[22] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware*, 2018.

[23] Kishori M. Konwar, Wyatt Lloyd, Haonan Lu, and Nancy A. Lynch. The SNOW theorem revisited. *CoRR*, abs/1811.10577, 2018.

[24] Diego Didona, Panagiota Fatourou, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. Distributed transactional systems cannot be fast. In *SPAA*, 2019.

[25] Karolos Antoniadis, Diego Didona, Rachid Guerraoui, and Willy Zwaenepoel. The impossibility of fast transactions. Technical Report 271320, EPFL, 2019.

[26] Christian Cachin, Rachid Guerraoui, and Luìs Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

[27] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. CreateSpace Independent Publishing Platform, 2017.

[28] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[29] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. From session causality to causal consistency. In *PDP*, 2004.

[30] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *VLDB*, 2013.

[31] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.

[32] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. Yesquel: Scalable sql storage for web applications. In *SOSP*, 2015.

[33] MySQL Cluster. https://www.mysql.com/products/cluster/. [Online; accessed 14-October-2019].

[34] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. 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. Spanner: Google's globally-distributed database. In *OSDI*, 2012.