The FIDS Theorems: Tensions Between Multinode and Multicore Performance in Transactional Systems

Naama Ben-David ✉
VMware Research, Palo Alto, California, USA

Gal Sela[1] ✉ 🔾
Technion, Haifa, Israel

Adriana Szekeres ✉
VMware Research, Bellevue, Washington, USA

──── **Abstract** ────

Traditionally, distributed and parallel transactional systems have been studied in isolation, as they targeted different applications and experienced different bottlenecks. However, modern high-bandwidth networks have made the study of systems that are both distributed (i.e., employ multiple nodes) and parallel (i.e., employ multiple cores per node) necessary to truly make use of the available hardware.

In this paper, we study the performance of these combined systems and show that there are inherent tradeoffs between a system's ability to have fast and robust distributed communication and its ability to scale to multiple cores. More precisely, we formalize the notions of a *fast deciding* path of communication to commit transactions quickly in good executions, and *seamless fault tolerance* that allows systems to remain robust to server failures. We then show that there is an inherent tension between these two natural distributed properties and well-known multicore scalability properties in transactional systems. Finally, we show positive results; it is possible to construct a parallel distributed transactional system if any one of the properties we study is removed.

1 Introduction

Transactional systems offer a clean abstraction for programmers to write concurrent code without worrying about synchronization issues. This has made them extremely popular and well studied in the last couple of decades [5, 21, 23, 41, 47, 53, 54, 48].

Many transactional systems in practice are *distributed* across multiple machines [14, 55, 30], allowing them to have many benefits that elude single-machine designs. For example, distributed solutions can scale to much larger data sets, handle much larger workloads, service clients that are physically far apart, and tolerate server failures. It is therefore unsurprising that distributed transactional systems have garnered a lot of attention in the literature, with many designs aimed at optimizing their performance in various ways: minimizing network round trips to commit transactions [29, 47, 54, 38, 15, 36], increasing robustness and availability when server failures occur [29, 54, 47], and scaling to heavier workloads [55, 19].

---

[1] Part of this work was done while this author was at VMware Research.

Due to increased bandwidth on modern networks, new considerations must be taken into account to keep improving the performance of distributed transactional systems. In particular, while traditional network communication costs formed the main bottleneck for many applications, sequential processing within each node is now no longer enough to handle the throughput that modern networks can deliver (through e.g., high-bandwidth links, multicore NICs, RDMA, kernel bypassing). Thus, to keep up with the capabilities of modern hardware, distributed transactional systems must make use of the parallelism available on each server that they use. That is, they must be designed while optimizing *both* network communication *and* multicore scalability.

Two main approaches have been employed by transactional storage systems to take advantage of the multicore architecture of their servers [46]: *shared-nothing* or *shared-memory*. The shared-nothing approach, where each core can access a distinct partition of the database and only communicates with other cores through message passing, has a significant drawback: cores responsible for hot data items become a throughput bottleneck while other cores are underutilized. To be able to adapt to workloads that stress a few hot data items, the shared memory approach, where each core can access any part of the memory, can be used. However, shared memory must be designed with care, as synchronization overheads can hinder scalability. Fortunately, decades of work has studied how to scale transactional systems in a multicore shared-memory setup [7, 6, 5, 10, 12, 40, 41, 8]. Thus, there is a lot of knowledge to draw from when designing distributed transactional systems that also employ parallelism within each server via the shared-memory approach.

In this paper, we study such systems, which we call *parallel distributed transactional systems (PDTSs)*. Our main contribution is to show that there is an inherent tension between properties known to improve performance in distributed settings and those known to improve performance in parallel settings. To show this result, we first formalize a model that combines both shared memory and message passing systems. While such a model has been formulated in the past [1], it has not been formulated in the context of transactional systems.

We then describe and formalize three properties of distributed transactional systems that improve their performance. These properties have all appeared in various forms intuitively in the literature [54, 29, 47], but have never been formalized until now. We believe that each of them may be of independent interest, as they capture notions that apply to many existing systems. In particular, we first present *distributed disjoint-access parallelism*, a property inspired by its counterpart for multicore systems, but which captures scalability across different distributed shards of data. Then, we describe a property that intuitively requires a *fast path* for transactions: transactions must terminate quickly in executions in which they do not encounter asynchrony, failures, or conflicts. While many fast-path properties have been formulated in the literature for consensus algorithms, transactions are more complex since different transactions may require a different number of network round trips, or message delays, in order to even know what data they should access. We capture this variability in a property we name *fast decision*, intuitively requiring that once the data set of a transaction is known, it must reach a decision within one network round trip. Finally, we present a property called *seamless fault tolerance*, which requires an algorithm to be able to tolerate some failures without affecting the latency of ongoing transactions. This has been the goal of many recent works which focus on robustness and high availability [47, 37, 38, 29, 54].

Equipped with these properties, we then show the inherent tension that exists between them and the well-known multicore properties of disjoint-access parallelism and invisible reads, both of which intuitively improve cache coherence and have been shown to increase scalability in transactional systems [42, 22]. More precisely, we present the **FIDS** theorem for

sharded PDTSs: a PDTS that guarantees a minimal progress condition and shards data across multiple nodes cannot simultaneously provide **F**ast decision, **I**nvisible reads, distributed **D**isjoint-access parallelism, and **S**erializability. An important implication of this result is that serializable shared-memory sharded PDTSs that want to provide multicore scalability cannot simply use a two-phase atomic commitment protocol (such as the popular two-phase commit). Furthermore, we turn our attention to *replicated* PDTSs. We discover that a similar tension exists for PDTSs that utilize *client-driven* replication. With client-driven replication replicas do not need to communicate with each other to process transactions. It is commonly used in conjunction with a leaderless replication algorithm to save two message delays [29, 54, 47, 38], as well as in RDMA-based PDTSs which try to bypass the replicas' CPUs [16, 44]. We present a *robust* version of the FIDS theorem, which we call the **R-FIDS** theorem: a PDTS (that may or may not shard its data) and utilizes client-driven replication cannot simultaneously provide **R**obustness to failures in the form of seamless fault tolerance, **F**ast decision, **I**nvisible reads, **D**isjoint-access parallelism, and **S**erializability.

Interestingly, similar impossibility proofs appear in the literature, often showing properties of parallel transactional systems that cannot be simultaneously achieved [41, 7, 12]. Indeed, some works have specifically considered disjoint-access parallelism and invisible reads, and shown that they cannot be achieved simultaneously with strong progress conditions [7, 41]. However, several systems achieve both disjoint-access parallelism and invisible reads with weak progress conditions such as the one we require [53, 48, 16]. To the best of our knowledge, the two versions of the FIDS theorem are the first to relate multicore scalability properties to multinode scalability ones.

Finally, we show that the FIDS theorems are minimal in the sense that giving up any one of these properties does allow for implementations that satisfy the rest.

In summary, our contributions are as follows.

- We present a transactional model that combines the distributed and parallel settings.
- We formalize three distributed performance properties that have appeared in intuitive forms in the literature.
- We present the FIDS and R-FIDS theorems for parallel distributed transactional systems, showing that there are inherent tensions between multicore and multinode scalability properties.
- We show that giving up any one of the properties in the theorems does allow designing implementations that satisfy the rest.

The rest of the paper is organized as follows. Section 2 presents the model and some preliminary notions. In Section 3, we define the properties of distributed transactional systems that we focus on. We present our impossibility results in Section 4, and then in Section 5, we show that it is possible to build a PDTS that sacrifices any one of the properties. Finally, we discuss related works in Section 6 and future research directions in Section 7.

2 Model and Preliminaries

Communication. We consider a *message-passing* model among *n nodes* (server hosts) and any number of *client processes*, as illustrated in Figure 1. Each node has *P node processes*. Messages are sent either between two nodes or between clients and nodes. We consider *partial synchrony* [18]; messages can be arbitrarily delayed until an a priori unknown *global stabilization time (GST)*, after which all messages reach their target within a known delay Δ. An execution is said to be *synchronous* if GST is at the beginning of the execution. Furthermore, node processes within a single node communicate with each other via *shared*
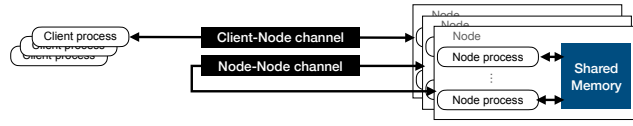
Figure 1 Communication mediums between the different types of processes considered in our model.

memory. That is, they access *shared base objects* through *primitive operations*, which are atomic operations, such as read, write, read-modify-write (compare-and-swap, test-and-set, fetch-and-increment, etc.), defined in the usual way. A primitive operation is said to be *non-trivial* if it may modify the object. Two primitive operations *contend* if they access the same object and at least one of them is non-trivial. The order of accesses of processes to memory is governed by a *fair scheduler* which ensures that all processes take steps.

Transactions.   We consider a database composed of a set of *data items*, Σ, which can be accessed by *read* and *write* operations. Each node N_i holds some subset $\Sigma_i \subseteq \Sigma$, which may overlap with the subsets held on other nodes. A *transaction T* is a program that executes read and write operations on a subset of the data items, called its *data set*, $D_T \subseteq \Sigma$. A transaction T's *write set*, $W_T \subseteq D_T$, and *read set*, $R_T \subseteq D_T$, are the sets of data items that it writes and reads, respectively. Two transactions are said to *conflict* if their data sets intersect at an item that is in the write set of at least one of them.

Transaction Interface.   An *application* may execute a transaction T by calling an *invokeTxn(T)* procedure. The invokeTxn(T) procedure returns with a *commit* or *abort* value indicating whether it committed or aborted, as well as the full read and write sets of T, with the order of execution of the operations (relative to each other), and with the read and written values. We say that a transaction is *decided* when invokeTxn(T) returns.

Failure Model.   Nodes can fail by crashing; if a node crashes then *all* processes on the node crash as well. We do not consider failures where individual processes crash and we assume clients do not fail. We denote by *failure-free* execution an execution without node crashes.

Handlers and Implementations.   An *implementation* of a PDTS provides data representation for transactions and data items, and algorithms for two types of handlers: the *coordinator handler* and the *message handler*. Each handler is associated with a transaction and is executed by a single process. Each process executes at most one handler at any given time, and is otherwise *idle*. The coordinator handler of a transaction T is the first handler associated with T and is triggered by an invokeTxn(T) call on some client process.

The execution of a handler involves a sequence of *handler steps*, which are of one of three types: (1) an *invocation* or *response* step, which is the first or last step of the handler respectively, (2) a primitive operation on a base object in shared memory, including its return value, and (3) sending or receiving a message, denoted send(T, m) or receive(T, m). Each handler step is associated with the corresponding transaction and the process that runs it. The return value in a response step of a transaction's coordinator handler is the return value of invokeTxn described above, and a message handler has no return value.

Executions.   An *execution* of a PDTS implementation is a sequence of handler steps and *node crash steps*. Each node crash step is associated with a node. After a node crash step associated with node N_i in execution E, no process on node N_i takes any steps in E. An execution can interleave handler steps associated with different transactions and processes. An *extension E'* of E is an execution that has E as its prefix.

We say that a transaction T's *interval* in an execution E begins at the invocation step of T's coordinator handler, and ends when there are no sends associated with T that have not been received whose target node has not crashed, and all handlers associated with T have reached their response step. Note that the end of a transaction's interval must therefore be a response step of some handler associated with T, but might not be the response step of T's coordinator handler (which may terminate earlier than some other handlers of T). We say that two transactions are *concurrent* in E if their intervals overlap. We say that two transactions, T_1 and T_2, *contend on node N_i* in E if they are concurrent, and there is at least one primitive operation step on node N_i in E associated with T_1 that contends with a primitive operation step in E associated with T_2. We say that T_1 and T_2 *contend* in E if there is some node N_i such that they contend on node N_i in E.

The *projection* of an execution E on a process p, denoted $E|p$, is the subexecution of E that includes exactly all of the steps associated with p in E. Two executions E and E' are *indistinguishable* to a process p if the projections of E and E' on p are identical (i.e., if $E|p = E'|p$).

It is also useful to discuss *knowledge* of properties during an execution. The notion of knowledge has been extensively used in other works [24, 20]. Formally, a process p *knows* a property P in an execution E of a PDTS implementation I, if there is no execution E' of I that is indistinguishable to p from E in which P is not true.

We adopt two concepts introduced by Lamport [35, 31] to aid reasoning about distributed systems: *depth of a step*, and the *happened-before* relation. The *depth of a step s* associated with transaction T in execution E is 0 if s is the invocation of T's coordinator handler. Otherwise, it equals the maximum of (i) the depths of all steps that are before s in E within the same handler as s, and (ii) if s is a receive(T, m) step of a message sent in a send(T, m) step, s', then 1 plus the depth of s'. *Happened-before* is the smallest relation on the set of steps of an execution E satisfying the following three conditions: 1) if a and b are steps of the same handler and a comes before b in E, then a *happened-before* b; 2) if a is a send(T, m) step and b is a receive(T, m) step, then a *happened-before* b; 3) if a *happened-before* b and b *happened-before* c, then a *happened-before* c.

Serializability.   Intuitively, a transactional system is *serializable* if transactions appear to have executed in some serial order [40]. The formal definition appears in the full version of this paper [9].

Weak Progress.   A transactional system must guarantee at least *weak progress*: every transaction is eventually *decided*, and every transaction that did not execute concurrently with any other transaction eventually *commits*.

2.1 Multicore Scalability Properties

To scale to many processes on each server node, transactional systems should reduce memory contention between different transactions. This topic has been extensively studied in the literature on parallel transactional systems [7, 6, 5, 11, 13, 23, 26, 41]. Here, we focus on two well-known properties, disjoint-access parallelism and invisible reads, that are known to reduce contention and improve scalability in parallel systems. We later show how they interact with distributed scalability properties.

2.1.1 Disjoint-Access Parallelism

Originally introduced to describe the degree of parallelism of implementations of shared memory primitives [26], and later adapted to transactional memory, *disjoint-access parallelism* intuitively means that transactions that are disjoint at a high level, e.g., whose data sets do not intersect, do not contend on shared memory accesses [7, 41]. While this property may sound intuitive, it can in fact be difficult to achieve, as it forbids the use of global locks or other global synchronization mechanisms. Multiple versions of disjoint-access parallelism exist in the literature, differing in which transactions are considered to be disjoint at a high level. In this paper, we use the following definition.

▶ **Definition 2.1** (Disjoint-access parallelism (DAP)). *An implementation of a PDTS satisfies* disjoint-access parallelism (DAP) *if two transactions whose data sets do not intersect cannot contend.*

2.1.2 Invisible Reads

The second property we consider, *invisible reads*, intuitively requires that transactions' read operations not execute any shared memory writes. This property greatly benefits workloads with read hotspots, by dramatically reducing cache coherence traffic. Two variants of this property are common in the literature. The first, which we call *weak invisible reads*, only requires invisible reads at the granularity of transactions. That is, if a transaction is read-only (i.e., its write set is empty), then it may not make any changes to the shared memory. This simple property has been often used in the literature [7, 41].

▶ **Definition 2.2** (Weak invisible reads). *An implementation of a PDTS satisfies* weak invisible reads *if, in all its executions, every transaction with an empty write set does not execute any non-trivial primitives.*

However, this property is quite weak, as it says nothing about the number of shared memory writes a transaction may execute once it has even a single item in its write set. When developing systems that decrease coherence traffic, this is often not enough. Indeed, papers that refer to invisible reads in the systems literature [47, 48] require that no read operation in the transaction be the cause of shared memory modifications. Note that an algorithm that locally stores the read set for validation (which is the case in the above referenced systems) can still satisfy invisible reads, since the writes are not to shared memory. Attiya et al. [6] formalize this stronger notion of invisible reads by requiring that we be able take an execution E and replace any transaction T in E with a transaction that has the same write set but an empty read set, and arrive at an execution that is indistinguishable from E. Intuitively, this captures the requirement that reads should not update shared metadata (e.g., through "read locks"). We adopt Attiya et al.'s definition of invisible reads here, adapted to fit our model.

▶ **Definition 2.3** (Invisible reads (adapted from [6])). *An implementation I of a PDTS satisfies the* invisible reads *property if it satisfies weak invisible reads and, additionally, for any execution E of I that includes a transaction T with write set W and read set R, there exists an execution E' of I identical to E except that it has no steps of T and it includes steps of a transaction T', which has the same interval as T (i.e., T's first and last steps in E are replaced by T''s ones in E'), and writes the same values to W in the same order as in T, but has an empty read set.*

Note that the invisible reads property complements the DAP property for enhanced multicore scalability. A system that has both allows all transactions that do not conflict, not just the disjoint-access ones, to proceed independently, with no contention (as we will

show in Lemma 4.4). Interestingly, previous works discovered some inherent tradeoffs of such systems [7, 41], in conjunction with strong progress guarantees. In this paper, we study these properties under a very weak notion of progress, but with added requirements on distributed scalability (see Section 3).

3 Multinode Performance Properties

To overcome the limitations of a single machine (e.g., limited resources, lack of fault tolerance), distributed transactional systems shard or replicate the data items on multiple nodes, and, thus, must incorporate distributed algorithms that coordinate among multiple nodes. The performance of these distributed algorithms largely depends on the number of communication rounds required to execute a transaction. Ideally, at least in the absence of conflicts, transactions can be executed in few rounds of communication, even if some nodes experience failures. In this section we propose formal definitions for a few multinode performance properties.

3.1 Distributed Disjoint-Access Parallelism

We start by proposing an extension of DAP to distributed algorithms, which we term *distributed-DAP*, or DDAP. In addition to requiring DAP, DDAP proscribes transactions from contending on a node unless they access common elements that reside at that node:

▶ **Definition 3.1** (Distributed disjoint-access parallelism (DDAP)). *An implementation of a PDTS satisfies* distributed disjoint-access parallelism (DDAP) *if for any two transactions T and T', and any node N_i, if T and T''s data sets do not intersect on node N_i (i.e., $D_T \cap D_{T'} \cap \Sigma_i = \emptyset$), then they do not contend on node N_i.*

While the main goal of sharding is to distribute the workload across nodes, DDAP links sharding to increased parallelism – DDAP systems can offer more node parallelism than DAP systems through sharding.

3.2 Fast Decision

Distributed transactional systems must integrate agreement protocols (such as *atomic commitment* and *consensus*) to ensure consistency across all nodes involved in transaction processing. Fast variants of such protocols can reach agreement in two message delays in "good" executions [35]. Ideally, we would like distributed transactional systems to preserve this best-case lower bound, and decide transactions in two message delays; reducing the number of message delays required to process transactions not only can significantly reduce the latency as perceived by the application (processing delay within a machine is usually smaller than the delay on the network), but can also reduce the *contention footprint* [21] of the transactions (intuitively, this is the duration of time in which a transaction might interfere with other transactions in the system).

Requiring transactions to be decided in just two message delays, however, is too restrictive in many scenarios. The latency of a distributed transactional system depends on how many message delays are required for a transaction to "learn" its data set (data items and their values); the data set needs to be returned to the application when the transaction commits, and is also used to determine whether the transaction can commit. For example, for interactive transactions or disaggregated storage, the values must be made available to the application (which runs in a client process) before the transaction can continue to execute. Thus, since

the data items are remote, each read operation results in two message delays, one to request the data from the remote node and one for the remote node to reply. For non-interactive transactions or systems where transaction execution can be offloaded to the node processes, the latency for learning the data set can be improved; since the client does not need to immediately know the return value of read operations, the values of data items can be learned through a chain of messages that continue transaction processing at the nodes containing the remote data. More precisely, the client first determines a node, n_1, that contains the first data item the transaction needs to read; the client sends a message to n_1 containing the transaction; n_1 processes the transaction, preforming the read locally, until it determines that the transaction needs to perform a remote read from another node, n_2; n_1 sends a message to n_2 containing the transaction and its state so far; n_2 continues processing the transaction, performing the read locally, and so on. RPC chains [45] already provides an implementation of this mechanism, saving one message delay per remote read operation. At the lowest extreme, non-sharded transactional systems can learn a transaction's entire data set in a single message delay.

We introduce the *fast decision* property to describe distributed transactional systems that can decide each transaction in "good" executions within only two message delays in addition to the message delays it requires to "learn" the transaction's entire data set. As explained above, the number of message delays required to learn a transaction's data set depends on several design choices. We note that often, deciding a transaction's outcome within two message delays after learning its data set is not plausible if the execution has suboptimal conditions, for example, if there are transactional conflicts that need to be resolved, or if not all nodes reply to messages within some timeout. This is true even for just consensus, where the two-message-delay decisions can happen only in favorable executions, on a *fast path* [3, 34]. We therefore define the fast decision property to only be required in such favorable executions.

To formalize fast decisions, we must be able to discuss several intuitive concepts more formally. In particular, we begin by defining the *depth of a transaction*, to allow us to formally discuss the number of message delays that the transactional system requires to decide a transaction.

▶ **Definition 3.2** (Depth of a transaction). *The* depth of a decided transaction T *in execution* E *of a PDTS implementation,* $d_E(T)$*, is the depth of the response step of T's coordinator handler in E.*

In many cases, we need to refer to the depth of a transaction T in an execution in which T is still ongoing, and its coordinator handler has not reached its response step yet. While we could simply refer to the depth of the deepest step of T in the execution, this would not be appropriate: it is possible that a transaction in fact took steps along one "causal path" that led to a large depth, but when the response step to T's coordinator handler happens, its depth is actually shorter. In such a case, we really only care about the depth along the "causal paths" that lead to the response step, since these are the ones affecting the latency to the application. To capture this notion, we define the *partial depth* of a transaction T in a prefix of an execution in which T is decided as follows.

▶ **Definition 3.3** (Partial depth of a transaction). *Let T be a decided transaction in execution E of a PDTS implementation. The* partial depth *of T in a prefix P of E in which T is not decided, $d_E(T, P)$, is the maximum step depth across all steps associated with T in P, which happened-before the response step of T's coordinator handler in E (or 0 if there are no such steps).*

We next formalize another useful concept that we need for the discussion of fast decisions; namely, what it means to *learn* the data set of a transaction. For that purpose, we introduce the following two definitions:

▶ **Definition 3.4** (Decided data item). *A data item d is* decided *to be in a transaction T's read or write set in execution E of a PDTS implementation if, in all extensions of E, the read or write set respectively in the return value of invokeTxn(T) contains d.*

▶ **Definition 3.5** (Decided value). *A data item d's value is* decided *for T in execution E of a PDTS implementation if, in all extensions of E, the read set in the return value of invokeTxn(T) contains d and with the same value.*

Note that a data item's value can be decided for a transaction only if that data item is part of its read set; the definition does not apply for data items in the write set. In the definition of the fast decision property and in the proofs, we refer in most places to knowing the decided values and not the data items in the write set as well. This is because knowing the read set and its values implies that a transaction's write set is decided in case it commits; this is the property that matters in many of the arguments we use in the paper.

Finally, we are ready to discuss the *fast decision* property. Intuitively, the formal definition of the property considers *favorable* executions, which are synchronous, failure-free and have each transaction run solo. For those executions, the property requires two things to hold: first, a transaction is not allowed to spend more than two message delays without learning some new value for its data set, and second, once its entire data set is known, it must be decided within 2 more message delays (Corollary 3.7). This captures "speed" in both learning the data set and deciding the transaction outcome. As discussed above, 2 message delays is an upper bound on the minimal amount of time needed to perform a read operation (and bring its value to the necessary process). Note that this is a tight bound for systems processing interactive transactions, and as such, fast decision also means optimal latency for these systems.

▶ **Definition 3.6** (Fast decision). *A PDTS implementation I is* fast deciding *if, for every failure-free synchronous execution E of I and every decided transaction T in E that did not execute concurrently with any other transaction, for any prefix P of E such that $d_E(T, P) < d_E(T) - 2$, there exists a prefix of E of partial depth $d_E(T, P) + 2$ in which the number of values known by some process to be decided is bigger than in P.*

Formalizing the allowed depth of a transaction in terms of prefixes of an execution in which the transaction is already decided (so we know its depth in that execution) helps capture the two requirements we want: (1) for any prefix of the execution, if we advance from it by two message delays, we must have improved our knowledge of the values of the read set, and (2) once the read set and its values are completely known (regardless of the depth of the prefix in which this occurs), we must be at most 2 message delays from deciding that transaction. Corollary 3.7 helps make this intuition concrete.

▶ **Corollary 3.7.** *For every failure-free synchronous execution E of a fast-deciding PDTS implementation I and every decided transaction T in E that did not execute concurrently with any other transaction, let P be the shortest prefix of E in which the value of each item in T's read set is known by some process to be decided. Then*

$$d_E(T) \leq d_E(T, P) + 2.$$

(Intuitively, T must be decided within at most 2 message delays from when T's read set including its values are known to be decided.)

Proof. Assume by contradiction that $d_E(T) > d_E(T, P) + 2$. Then by the fast decision property of I, there exists a prefix of E in which the number of data items whose value is known by some process to be decided is bigger than in P. But this is impossible, since the values of T's entire read set are known to be decided in P.   ◀

Several fast-deciding distributed transactional systems have been recently proposed for general interactive transactions [29, 54, 47]; our fast decision property captures what they informally refer to as "one round-trip commitment". These systems use an *optimistic concurrency control* and start with an execution phase that constructs their data sets with two message delays per read operation. The agreement phase consists of validation checks that require a single round-trip latency (integrates atomic commitment and a fast consensus path in one single round trip). The write phase happens asynchronously, after the response of the transaction has been emitted to the application.

3.3   Seamless Fault Tolerance

High availability is critical for transactional storage systems, as many of their applications expect their data to be always accessible. In other words, the system must mask server failures and network slowdowns. To achieve this, many systems in practice are designed to be fault tolerant; the system can continue to operate despite the failures of some of its nodes.

However, oftentimes, while the system can continue to function when failures occur, it experiences periods of unavailability, or its performance degrades by multiple orders of magnitude while recovering [3, 51]. This is the case in systems that must manually reconfigure upon failures [50], and those that rely on a leader [3, 39, 51, 33, 32].

These slow failure-recovery mechanisms, while providing some form of guaranteed availability, may not be sufficient for systems in which high availability is truly critical; suffering from long periods of severe slowdowns potentially from a single server failure may not be acceptable in some applications.

To address this issue, some works in recent years have focused on designing algorithms that experience minimal slowdowns, or no slowdowns at all, upon failures. One approach has been to minimize the impact of leader failures by making the leader-change mechanism lightweight and switching leaders even when failures do not occur [52]. Another approach aims to eliminate the leader completely; such algorithms are called *leaderless* algorithms [4, 47, 54, 37]. All of these approaches aim to tolerate the failure of some nodes without impacting the latency of ongoing transactions.

In this paper, we formalize this goal of tolerating failures without impacting latency into a property that we call *s-seamless fault tolerance*, where $s \leq f$. In essence, s-seamless fault tolerance requires that if only up to s failures occur in an execution, no slowdown is experienced. To capture this formally, we require that for any execution E with up to $s - 1$ crashes, it be possible to find an equivalent execution E' with one more crash event, which may happen at any time after the crashes in E, where the depth of all transactions are the same in E and E'. We express this in an inductive definition.

▶ **Definition 3.8** (*s*-seamless fault tolerance). *Any implementation of a PDTS satisfies 0-seamless fault tolerance. An implementation I of a PDTS satisfies s-seamless fault tolerance if it satisfies $(s - 1)$-seamless fault tolerance, and for any execution E of I with $s - 1$ node crashes, for any prefix E_P of E that contains the $s - 1$ node crashes, and any node crash event c of a node that has not crashed in E_P, there exists an execution E' of I whose prefix is $E_P \cdot c$, such that (1) stripping each of E and E' of all steps other than invocation and response steps of coordinator handlers results in the same sequence of invocation and response steps (intuitively, the executions are equivalent), and (2) the depth of each decided transaction is the same in both executions (intuitively, E' seamlessly tolerates the node crashes).*

While *s*-seamless fault tolerance offers the extremely desirable robustness property, it also requires that: a) no single node can be on the critical path of all transactions, and b) no single node can be solely responsible for processing a transactional task. This can be a double-edged sword; on the one hand, this eliminates the possibility of a leader bottleneck, which implies better scalability. On the other hand, it disallows certain optimizations, like reading from a single replica.

4 Impossibility Results

Having specified some key properties which make distributed transactional systems fast and scalable, we now turn to the main result of our paper: unfortunately, there is a tension between these multinode performance properties and the single-node multicore performance properties discussed in Section 2. More specifically, we present the **FIDS** theorems, which formalize the impossibility of achieving all of these properties simultaneously in two different parallel distributed settings.

4.1 The FIDS Theorems

The first **FIDS** theorem states that no PDTS with weak progress which *shards data* can guarantee **F**ast decision, **I**nvisible reads, distributed **D**isjoint-access parallelism, and **S**erializability simultaneously. This is in contrast to known systems that achieve just the multinode properties [47, 54, 38] or just the multicore properties [53, 48, 16]. Thus, the FIDS theorem truly shows tensions that arise when a transactional system is both parallel and distributed. This version of the FIDS theorem considers only systems that shard data, that is, systems in which each node only stores part of the database items. Interestingly, the impossibility holds in this setting even without requiring any fault tolerance, and in particular, without seamless fault tolerance. We note that the FIDS theorem applies also to systems that replicate data in addition to sharding it; adding replication on top of a sharded system only makes it more complex. Formally:

▶ **Theorem 4.1** (The FIDS theorem for sharded transactional systems). *There is no implementation of a PDTS which shards data across multiple nodes that guarantees weak progress, and simultaneously provides fast decision, invisible reads, distributed disjoint-access parallelism, and serializability.*

For systems that maintain multiple copies of the data, but do not necessarily shard it, we show a different version of the result. Note that in such systems, distribution comes from replication; several nodes, each with a copy of the entire database, are used to ensure fault tolerance. For this setting, we present the Robust-FIDS, or **R-FIDS**, theorem: a PDTS with weak progress that utilizes client-driven replication and satisfies **R**obustness to at least one failure through the seamless fault tolerance property, in addition to satisfying **F**ast decision, **I**nvisible reads, **D**isjoint-access parallelism, and **S**erializability, is also impossible to implement. Formally:

▶ **Theorem 4.2** (The R-FIDS theorem for replicated transactional systems). *There is no implementation of a PDTS that utilizes client-driven replication that guarantees weak progress, and simultaneously provides 1-seamless fault tolerance, fast decision, invisible reads, disjoint-access parallelism, and serializability.*

In the reminder of this section we present an overview of the proof technique for the two versions of the FIDS theorem; the detailed proof for each of them and the supporting lemmas we introduce here can be found in Appendix A.

4.2   Proof Overview

Both proofs have a similar structure; we consider example transactions that form a dependency cycle, and show an execution in which all of them commit, thereby violating serializability. To argue that all transactions in our execution commit, we build the execution by merging executions in which each transaction ran solo (and therefore had to commit by weak progress), and showing that the resulting concurrent execution is indistinguishable to each transaction from its solo run. Starting with solo executions also gives us another property that we can exploit; we define the solo executions to be synchronous and failure-free, and therefore they must be fast deciding as well.

The key challenge in the proofs is how to construct a concurrent execution E_{concur} that remains indistinguishable to all processes from the solo execution that they were a part of. To do so, we divide the concurrent execution into two phases; first, we let the solo executions run, in any interleaving, until right before the point in each execution at which some process learns the values of its transaction's read set. When this point is reached in each solo execution, we carefully interleave the remaining steps in a second phase of the concurrent execution. A key feature is that by the fast decision property, which each solo execution satisfies, once some process learns the read set including its values, there are at most two message delays left in each solo execution before the transaction is decided. This bounds the amount of communication we need to worry about in the second phase of the concurrent execution.

To show that E_{concur} is indistinguishable from the solo runs, we look at each of the two phases separately. The idea is to show that no process makes *any* shared memory modifications in the first phase, and then show that we can interleave messages and message handlers in a way that allows each transaction to be oblivious to the other transactions for at least one more intuitive "round trip", which is all we need to reach decision according to the fast decision property.

To show that a transaction performs no shared memory modifications in the first phase of the concurrent execution we construct, we rely on the way we choose the transactions, their data sets, and when in the execution their data sets are decided; in both proofs, the transactions we choose may have empty or non-empty write sets, depending on the results of their reads. The following lemma shows that as long as a transaction's write set is not *known* to be non-empty, the transaction cannot cause any modifications in a system that provides weak invisible reads.

▶ **Lemma 4.3.** *Let I be an implementation of a PDTS that provides weak invisible reads, and let T be a transaction in an execution E of I, such that no process in E knows the following proposition: T's write set is non-empty in all extensions of this execution in which T is decided. Then T cannot cause any base object modifications in E.*

This lemma, combined with the way we choose the transactions in our proofs, immediately implies that phase 1 of E_{concur} is indistinguishable to all processes from the solo executions they are a part of.

The proofs differ somewhat in how they show that E_{concur} is indistinguishable from the solo runs in the second phase. We argue about restricted shared memory modifications through the use of the DAP and invisible reads properties in the following key lemma, which intuitively shows that transactions that do not conflict do not (visibly) contend.

▶ **Lemma 4.4.** *Let I be an implementation of a PDTS that provides both DAP and invisible reads, and let T be a transaction in an execution E of I, such that its final write set is W. Then T does not cause any base object modifications visible to any concurrent transaction in E whose data set does not overlap with W.*

To make phase 2 of E_{concur} also indistinguishable from the solo executions, we schedule the remaining messages carefully. In particular, we schedule messages sent by reading transactions to each node before those sent by writing transactions, and again rely on DAP and invisible reads to argue that the reading transactions' handlers will not cause changes visible to those who write afterwards. However, here the two proofs diverge.

4.2.1 Sharded Systems

We first discuss the proof structure for showing that serializable sharded transactional systems that provide weak progress cannot simultaneously achieve fast decision, invisible reads and DDAP. That is, sharding the data across multiple nodes while achieving these properties is impossible even if we do not tolerate any failures (Theorem 4.1).

The proof uses two nodes and two transactions, each reading from a data item on one node and, if it sees the initial value, writing on the other node. The read set of one transaction is the same as the (potential) write set of the other transaction. We need to argue that the reading transaction on some node cannot cause modifications on that node that are visible to the writing transaction. However, since the write set of each transaction overlaps with the data set of the other, we cannot apply Lemma 4.4. Instead, we rely on DDAP, and show that with this property, the reading transaction indeed cannot be visible to the writing one on each node. We show a lemma very similar to Lemma 4.4 but which applies to transactions whose write set on a specific node does not overlap the data set of another transaction on that node.

▶ **Lemma 4.5.** *In any implementation of a PDTS that provides both DDAP and invisible reads, a transaction whose write set is W does not cause any modifications on shared based objects on a node N visible to any concurrent transaction whose data set does not overlap with W on N.*

The proof of this lemma is very similar to the proof of Lemma 4.4. The only required adjustments are using $DDAP$ instead of DAP, and referring to T''s data set and T's write set and modification *on a certain node N*.

Note that while the proof of the FIDS theorem relies on sharding, it does not need fault tolerance. In particular, it does not make use of the seamless fault tolerance property. However, the result does apply to systems in which the data is both sharded and replicated, as those systems are even more complex than ones in which no replication is used.

4.2.2 Replicated but Unsharded

So far, we have considered a PDTS in which node failures cannot be tolerated; if one of the nodes crashes, we lose all data items stored on that node, and cannot execute any transactions that access those data items. However, in reality, server failures are common, and therefore many practical systems use replication to avoid system failures. Of course, the impossibility result of Theorem 4.1 holds for a PDTS even for the more difficult case in which failures are possible and each node's data is replicated on several backups.

However, we now turn our attention to PDTSs in which the entire database is stored on each node. This setting makes it plausible that a client could get away with accessing only one node to see the state of the data items of its transaction. However, we show that the impossibility of Theorem 4.1 still holds in this setting for a system in which failures are tolerated without affecting transaction latency (i.e., systems that satisfy seamless fault tolerance) (Theorem 4.2).

As explained in Section 4.2, the use of seamless fault tolerance requires us to explicitly argue about the length of the executions in which transactions decide. To do so, we need the following lemma, which gives a lower bound for the depth at which a transaction's read set and values can be decided.

▶ **Lemma 4.6.** *There is no execution E of any serializable PDTS implementation that tolerates at least 1 failure in which there is a transaction T and prefix P such that $d_E(T, P) < 2$ and some process knows the decided value of some read of T in P.*

Once we have this lemma, the proof of the R-FIDS theorem is then similar to the proof of the FIDS theorem. We build a cycle of dependencies between transactions where each neighboring pair in the cycle overlaps on a single data item that one of them reads and the other writes. The key is that because of invisible reads, each read can happen before the write on the same data item without leaving a trace. However, to construct this cycle in the replicated case, we need at least 3 replicas, 3 transactions and 3 data items. This is because we can no longer separate the read and write of a single transaction on each node. Furthermore, we make use of Lemma 4.6, as well as the budgeted depth of a transaction in a fast-deciding execution, to explicitly argue about the amount of communication possible after a transaction learns its write set.

More specifically, we choose three transactions, where the write set of one equals the read set of the next. We divide them into pairs, where within each pair, the write set of one does not overlap with the data set of the other. We can then directly use Lemma 4.4 to argue that the second one to be scheduled of this pair will not see changes made by the first. We exploit fault tolerance to have the third transaction's messages never reach that node. However, here, we must be careful, since we defined the solo executions to be failure-free to guarantee fast decisions. We therefore rely on seamless fault tolerance; we show indistinguishability of the concurrent execution not from the original solo executions, but from executions of the same depth that we know exist due to seamless fault tolerance.

Interestingly, when we convert a solo execution S to an execution F of the same depth (but with a node failure) via the seamless fault tolerance property, we may lose its fast decision property. That is, while the new execution must have the same depth as the original ones, that does not guarantee that it will also be fast deciding, as the fast decision property does not solely refer to the length of the execution. In particular, it could be the case that in F, the data set of a transaction including its values is learned earlier, but then the transaction takes more than 2 message delays to be decided. This would be problematic for our proof, in which the indistinguishability relies heavily on fast decision once the data set including its values are known. To show that this cannot happen in the executions we consider, we rely on Lemma 4.6 that bounds the depth at which any transaction in a fault tolerant system can learn the decided values of its reads.

5   Possibility Results

Any subset of the properties outlined in Theorem 4.2 *is* possible to achieve simultaneously in a single system. Due to lack of space, we show this in the full version of this paper [9], where we present four distributed transactional system algorithms, each sacrificing one of the desired properties. Recall from our model description that the presented protocols work under the assumption that the client does not fail and nodes do not recover, and as such are not intended to be used "as is" in practice. We first present a "base" algorithm which achieves all the desired properties (i.e., fast decision, invisible reads, DDAP, and

1-seamless fault tolerance), but is not serializable. We obtain each of the four transactional systems algorithms, by tweaking the base algorithm to sacrifice one desired property and gain serializability.

6 Related Work

Disjoint-access parallelism was first introduced in [26] in the context of shared memory objects. It was later adapted to the context of transactions. Over the years, it has been extensively studied as a desirable property for scalable multicore systems [53, 49, 47, 7, 5, 41, 23]. Several versions of DAP have been considered, differing in what is considered a conflict between operations (or transactions). A common variant of DAP considers two transactions to conflict if they are connected in the conflict graph of the execution (where vertices are transactions and there is an edge between two transactions if their data sets intersect) [7, 41]. In this paper, we consider a stricter version, which only defines transactions as conflicting if they are neighbors in the conflict graph. This version has also appeared frequently in the literature [41, 12].

Invisible reads have also been extensively considered in the literature [43, 7, 47, 49, 53, 25]. Many papers consider invisible reads on the granularity of data item accesses; any read operation on a data item should not cause changes to shared memory [49, 47, 25]. Others, often those that study invisible reads from a more theoretical lens, consider only the invisibility of *read-only transactions* [7, 41].

Some impossibility tradeoffs for transactional systems, similar to the one we show in this paper, are known in the literature. Attiya et al. [7] show that it is impossible to achieve weak invisible reads, disjoint-access parallelism, and wait-freedom in a parallel transactional system. Peluso et al. [41] show an impossibility of a similar setting, with disjoint-access parallelism, weak invisible reads, and wait-freedom, but consider any correctness criterion that provides real-time ordering. Bushkov et al. [12] show that it is impossible to achieve disjoint-access parallelism and obstruction-freedom, even when aiming for consistency that is weaker than serializability. In this paper, none of our algorithms provide the obstruction-freedom considered in [12]; we use locks, and our algorithms can therefore indefinitely prevent progress if process failures can occur while holding locks.

Fast paths for *fast decision* have been considered extensively in the replication and consensus literature [28, 17, 34, 2, 3]. In most of these works, the conditions for remaining on the fast path include experiencing no failures. That is, they do not provide seamless fault tolerance. However, some algorithms, like Fast Paxos [34], can handle some failures without leaving the fast path. In the context of transactional systems, the fast path is often considered for conflict-free executions rather than those without failures [47, 54, 38, 29], as we do in this paper. Seamless fault tolerance captures the idea that (few) failures should not cause an execution to leave the fast path. Systems often have a general fault tolerance f that is higher than the number of failures they can tolerate in a seamless manner [47, 54, 38, 29].

Seamless fault tolerance as presented in this paper is also related to *leaderlessness* [37, 4], as any leader-based algorithm would slow down upon a leader failure. However, the leaderless requirement alone is less strict than our seamless fault tolerance; Antoniadis et al. [4] defined a leaderless algorithm as any algorithm that can terminate despite an adaptive adversary that can choose which process to temporarily remove from an execution at any point in time. This does not put any requirement on the speed at which the execution must terminate.

Parallel distributed transactional systems have been recently studied in the systems literature. Meerkat [47] provides serializability and weak progress, and three of the desirable properties we outline in this paper. It does not, however, provide invisible reads in any form

(not even the weaker version). Eve [27] considers replication for multicore systems. It briefly outlines how PDTS transactions are possible using its replication system, but it is not their main focus.

7   Discussion

This paper is inspired by recent trends in network capabilities, which motivate the study of distributed transactional systems that also take advantage of the parallelism available on each of their servers. We formalize three performance properties of distributed transactional systems that have appeared intuitively in various papers in the literature, and show that these properties have inherent tensions with multicore scalability properties. In particular, in this paper we formalized the notions of distributed disjoint-access parallelism, a fast decision path for transactions, and robustness in the form of seamless fault tolerance. Combined with the well-known multicore scalability properties of disjoint-access parallelism and invisible reads, we show the *FIDS theorem*, and its fault tolerant version, the *R-FIDS theorem*, which show that serializable transactional systems cannot satisfy all these properties at once. Finally, we show that removing any one of these properties allows for feasible implementations.

We note that our possibility results can be seen as "proofs of concept" rather than practical implementations. It would be interesting to design practical algorithms that give up just one of the properties we discuss. We believe that each property has its own merit for certain applications and workloads, and it would be interesting to determine which property would be the best to abandon for which types of applications.

In this work, we focused on studying parallel distributed transactional systems under a minimal progress guarantee. It would also be interesting to explore PDTSs under stronger progress conditions, or consistency conditions other than serializability. It would be equally interesting to see if the tension still exists between weaker variants of the properties we considered.

References

1   Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 51–60, 2018.

2   Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

3   Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616, 2020.

4   Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 392–402. IEEE, 2021.

5   Hagit Attiya and Panagiota Fatourou. Disjoint-access parallelism in software transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 72–97. Springer, 2015.

6   Hagit Attiya and Eshcar Hillel. The cost of privatization in software transactional memory. *IEEE Transactions on Computers*, 62(12):2531–2543, 2012.

7   Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

8 Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In *Colloquium on Structural Information & Communication Complexity*, 2008.

9 Naama Ben-David, Gal Sela, and Adriana Szekeres. The FIDS Theorems: Tensions between Multinode and Multicore Performance in Transactional Systems, 2023. `arXiv:2308.03919`.

10 Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

11 Silas Boyd-Wickizer. *Optimizing communication bottlenecks in multiprocessor operating system kernels*. PhD thesis, Massachusetts Institute of Technology, 2014.

12 Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. The pcl theorem: Transactions cannot be parallel, consistent and live. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 178–187, 2014.

13 Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. Association for Computing Machinery.

14 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. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

15 James Cowling and Barbara H. Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, pages 223–236, 2012.

16 Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2815400.2815425`.

17 Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 22–27. IEEE, 2005.

18 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

19 Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

20 Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Vardi. *Reasoning about knowledge*. MIT press, 2004.

21 Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD '14*, 2014.

22 Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

23 Rachid Guerraoui and Michal Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 304–313, 2008.

24 Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

25 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.

26 Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/197917.198079`.

27 Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, USA, 2012. USENIX Association.

28 Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2):45–63, 2001.

29 Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2465351.2465363`.

30 Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. `doi:10.1145/1773912.1773922`.

31 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. `doi:10.1145/359545.359563`.

32 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

33 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

34 Leslie Lamport. Fast paxos. *Distributed Comput.*, 19(2):79–103, 2006. `doi:10.1007/s00446-006-0005-x`.

35 Leslie Lamport. Lower bounds for asynchronous consensus. *Distrib. Comput.*, 19(2):104–125, October 2006. `doi:10.1007/s00446-006-0155-x`.

36 Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, SOSP '17, 2017.

37 Iulian Moraru, David G Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

38 Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 517–532, USA, 2016. USENIX Association.

39 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

40 Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

41 Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 217–226, 2015.

42 Amitabha Roy, Steven Hand, and Tim Harris. Exploring the limits of disjoint access parallelism. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA*, 2009.

43 William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, 2005.

44 Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *SIGMOD'19*, 2019.

45 Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. Rpc chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

46 Michael Stonebraker. The case for shared nothing. In *IEEE Database Eng. Bull.*, 1985.

47 Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

48 Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2517349.2522713`.

49 Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

50 Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

51 Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

52 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

53 Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, 2016.

54 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2815400.2815404`.

55 Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*, 2021.

Appendix

Full proofs of the FIDS theorems (and the supporting lemmas)

▶ **Lemma 4.3.** *Let I be an implementation of a PDTS that provides weak invisible reads, and let T be a transaction in an execution E of I, such that no process in E knows the following proposition: T's write set is non-empty in all extensions of this execution in which T is decided. Then T cannot cause any base object modifications in E.*

Proof. Let I be an implementation of a PDTS that satisfies weak invisible reads. Assume by contradiction that there is a transaction T in execution E of I, such that no process in E knows that T's final write set is not empty in all extensions in which T is decided, and a process p runs a handler associated with T that performs some base object modification.

Since p does not know that T's final write set is not empty in all extensions of the current execution in which T is decided, there exists an execution indistinguishable to p from E that has an extension in which T's final write set is empty. Let that extension be $E_{readOnly}$. Since T's final write set in $E_{readOnly}$ is empty, then by weak invisible reads, T cannot cause base object modifications in $E_{readOnly}$. Contradiction. ◀

▶ **Lemma 4.4.** *Let I be an implementation of a PDTS that provides both DAP and invisible reads, and let T be a transaction in an execution E of I, such that its final write set is W. Then T does not cause any base object modifications visible to any concurrent transaction in E whose data set does not overlap with W.*

Proof. Let I be an implementation of a PDTS that satisfies DAP and invisible reads. Let T be a transaction whose final write set in an execution E of I is W. Assume by contradiction that there exists some transaction T' concurrent with T in E whose data set does not overlap with W, but which sees a modification made by T in E. That is, there is some base object operation step s of T' whose return value is affected by T's modification.

By invisible reads, there exists an execution E' of I identical to E except that it includes a transaction T_{noRead} in place of T with the same interval, where T_{noRead} has W as its write set and an empty read set. By DAP, T_{noRead} does not modify in E' any base object accessed by any concurrent transaction whose data set does not overlap with W. In particular, T_{noRead} cannot make any modifications visible to T' in E'. Note that step s must exist in E', since by definition, E' is identical to E except in steps associated with T and T_{noRead}. However, in E, s's return value is affected by T's modification, and in E', this modification does not exist. Therefore, E' cannot be an execution of I. Contradiction. ◀

▶ **Lemma 4.5.** *In any implementation of a PDTS that provides both DDAP and invisible reads, a transaction whose write set is W does not cause any modifications on shared based objects on a node N visible to any concurrent transaction whose data set does not overlap with W on N.*

Proof. Let I be an implementation of a PDTS that satisfies DDAP and invisible reads. Let T be a transaction whose final write set on a node N in an execution E of I is W. Assume by contradiction that there exists some transaction T' concurrent with T in E whose data set on N does not overlap with W, but which sees a modification made by T on a base object on N in E. That is, there is some base-object operation step s of T' whose return value is affected by T's modification.

By invisible reads, there exists an execution E' of I identical to E except that it includes a transaction T_{noRead} in place of T with the same interval, where T_{noRead} has W as its write set on N and an empty read set. By DDAP, T_{noRead} does not modify in E' any base object

on N accessed by any concurrent transaction whose data set does not overlap with W on N. In particular, T_{noRead} cannot make any modifications visible to T' on N in E'. Note that step s must exist in E', since by definition, E' is identical to E except in steps associated with T and T_{noRead}. However, in E, s's return value is affected by T's modification, and in E', this modification does not exist. Therefore, E' cannot be an execution of I. Contradiction. ◀

▶ **Lemma 4.6.** *There is no execution E of any serializable PDTS implementation that tolerates at least 1 failure in which there is a transaction T and prefix P such that $d_E(T, P) < 2$ and some process knows the decided value of some read of T in P.*

Proof. Assume by contradiction that there is some implementation I of a serializable PDTS that tolerates at least 1 failure, an execution E of I, and a prefix P of E such that $d_E(T, P) \leq 1$ and some process knows the decided value of some data item d of T in P. Without loss of generality, let process p on node N be the process that knows d's decided value, let that value be v and let T's invoking client be C. Note that since C does not have access to the data, and any step of any process not on C's node must be of depth at least 1, p cannot be on C's node, and cannot have received any message from any process other than C within depth less than 2. Therefore p can only know the value of d on node N, but not any other nodes. Consider the following executions.

E_{N-fail}. E_{N-fail} and E are identical up to right before T's invocation. In E_{N-fail}, node N fails at this point. Then, a transaction T' is invoked by a client $C' \neq C$. T' writes a value $v' \neq v$ to d and commits. After T' commits, T is invoked in E_{N-fail}. Clearly, by serializability, T's read of d in E_{N-fail} returns v' or a more updated value, but not v.

E_{N-slow}. E_{N-slow} is identical to E_{N-fail} except that node N does not fail in E_{N-slow}. Instead, all messages to and from N are arbitrarily delayed in E_{N-slow} starting at the same point at which N fails in E_{N-fail}. Clearly, E_{N-slow} is indistinguishable from E_{N-fail} to all processes not on N.

E'. E' is identical to E_{N-slow} except that node N receives messages from client C. Clearly, E' and E_{N-slow} are indistinguishable to all processes not on N. So, T's read of d must return the same value as in E_{N-slow}, namely v' or a more updated one, but not v. However, note that E' is also indistinguishable to processes on N from E in any prefix of E of partial depth < 2 for T, since no process in N received any messages other than those it received in E, and since clients do not receive any messages not related to their own transactions, so C must have sent the same message(s) to N in E' as it did in E. Therefore, there is a prefix P' of E' indistinguishable to p from P, in which v is not the decided value of d, contradicting p's knowledge of d's decided value in P. ◀

▶ **Theorem 4.1** (The FIDS theorem for sharded transactional systems). *There is no implementation of a PDTS which shards data across multiple nodes that guarantees weak progress, and simultaneously provides fast decision, invisible reads, distributed disjoint-access parallelism, and serializability.*

Proof. Assume by contradiction that there exists an implementation I of a PDTS with all the properties in the theorem statement. Consider a database with 2 data items, X_1, X_2, partitioned on 2 nodes, N_1, N_2 respectively. Consider two transactions, T_1, T_2, with the following data sets: T_1's read set is $\{X_1\}$. Its write set is $\{X_2\}$ if its read returns the initial value of X_1, in which case it writes a value different from X_2's initial value. Otherwise, its write set is empty. For T_2, its read set is $\{X_2\}$, and its write set is $\{X_1\}$ if its read returns the initial value of X_2, and empty otherwise. If its write set is non-empty, it writes a value different from X_1's initial value. Let T_1 be executed by a client C_1 and T_2 be executed by a different client C_2. Consider the following executions.

Solo Executions.   We define two executions S_1, S_2, corresponding to T_1, T_2 respectively running in isolation, without the other transaction present in the execution. Both executions are synchronous and failure-free. By weak progress, T_i commits in S_i, and by serializability, T_i returns the initial value of its read item and therefore its write set is not empty.

Concurrent Execution.   We define an execution, E_{concur}, where T_1 and T_2 execute concurrently. On each node, each transaction is executed on different processes. Recall that this can happen since this is a parallel system, and the executing processes for a transaction are arbitrarily chosen among the idle processes of each node. In E_{concur}, for each transaction T_i, we let each process that executes it run until right before it knows the decided read set and read set value of T_i. Let the prefix of E_{concur} that includes all these steps be P_1. We then let each process that handles T_i run until when the next step of its handler has depth $\geq d_{S_i}(T_i) - 2$. Next, we let all messages sent on behalf of T_1 to N_1 and not yet received reach N_1 and be handled before any message sent on behalf of T_2 to N_1. For node N_2, we let the reverse happen; messages sent on behalf of T_2 reach it and are handled before messages sent on behalf of T_1. Finally, we resume all processes, and pause node processes that handle T_i when the next step of their handler has depth $\geq d_{S_i}(T_i)$. As for the client of each transaction, we let any messages sent to it arrive in the same order as they did in their corresponding solo executions (we will show that it receives the same messages in E_{concur}).

We now claim that execution E_{concur} is indistinguishable to C_i from S_i, and indistinguishable to each node process running T_i from the prefix of S_i containing all this process's steps of depth $< d_{S_i}(T_i)$. To do so, we consider the execution in two phases; the phase before the two transactions achieve knowledge of their data sets including their values (up to the end of P_1), and the phase afterwards.

Phase 1 of E_{concur}.   Note that for any prefix P of E_{concur} in which T_i's read set's value is not known to be decided by some process, T_i's known decided write set in P is empty. Consider the longest prefix $P_{undecided_i}$ of P_1 in which the decided write set of T_i is still empty. Note that for every process p, its knowledge of T_i's write set in $P_{undecided_i}$ is the same as it is in P. Therefore, by Lemma 4.3, in any such prefix P, T_i may not make any modifications to shared base objects visible to *any* concurrent transaction. Therefore, in phase 1 there are no modifications visible to either transaction that were not visible in the solo execution as well. Thus, by the end of phase 1, E_{concur}'s prefix P_1 is indistinguishable to both transactions from their respective solo executions. Therefore, both transactions read the initial values of their respective read sets, and both have a non-empty write set in E_{concur}.

Phase 2 of E_{concur}.   To show that E_{concur} remains indistinguishable from the solo executions to their respective transactions in phase 2, we rely on the order of messages that are received by the two nodes.

First, we note that by Lemma 4.5, T_i does not make base object modifications visible to $T_{(i \bmod 2)+1}$ on node N_i, since T_i's final write set is $\{X_{(i \bmod 2)+1}\}$, which does not intersect $T_{(i \bmod 2)+1}$'s final data set on node N_i.

Next, note that in each solo execution S_i, the first process that knows the decided value of T_i must be on node N_i, since that is where the data for the read of T_i is stored. Furthermore, by construction of E_{concur}, any messages sent on behalf of T_i to N_i immediately after both transactions gain knowledge of their write sets arrives before any such message sent on behalf of $T_{(i \bmod 2)+1}$, and its handler is completely executed. Thus, by the above claim, on both nodes, all handlers of both transactions for messages sent at depth $d_{E_{concur}}(T_i, P_1)$ execute to completion in a way that is indistinguishable to T_i from the solo execution S_i.

Finally, note that since S_i is synchronous and failure and conflict free, and I satisfies the fast decision property, by Corollary 3.7, the depth of T_i in S_i is at most 2 more than the partial depth of the first prefix in which T_i's data set including its values became known. In particular, since S_i is indistinguishable to processes executing T_i from E_{concur} up to that point, this means that $d_{S_i}(T_i) \le d_{E_{concur}}(T_i, P_1) + 2$. Thus, once messages from within the handlers that were activated by messages sent in E_{concur} at depth $d_{E_{concur}}(T_i, P_1)$ are received, T_i must be decided in E_{concur} as well, since E_{concur} is indistinguishable from S_i to all processes running T_i up to this point. Therefore, both transactions commit successfully in E_{concur} in a manner indistinguishable from their respective solo executions.

However, this yields a circular dependency between the two transactions; T_2 must occur before T_1, since it returns the initial value of X_2, before T_1 writes to it. Similarly, T_1 must occur before T_2, since it returns the initial value of X_1. This therefore contradicts serializability. ◀

▶ **Theorem 4.2** (The R-FIDS theorem for replicated transactional systems). *There is no implementation of a PDTS that utilizes client-driven replication that guarantees weak progress, and simultaneously provides 1-seamless fault tolerance, fast decision, invisible reads, disjoint-access parallelism, and serializability.*

Proof. Assume by contradiction that there exists an implementation I of a parallel replicated transactional system with all the properties stated in the theorem.

Consider a transactional system with 3 nodes N_1, N_2, N_3. (For less than 3 nodes, there is no PDTS that tolerates $f \ge 1$ failures in the partial-synchrony model [18].) Further consider 3 transactions T_1, T_2, T_3, 3 client processes C_1, C_2, C_3, and 3 data items X_1, X_2, X_3 each of which is replicated on all 3 nodes. The data sets of the transactions are as follows: T_i's read set includes $X_{(i \bmod 3)+1}$, and if the result of T_i's read of $X_{(i \bmod 3)+1}$ is the initial value of $X_{(i \bmod 3)+1}$, its write set includes X_i. Otherwise, its write set is empty. Each transaction T_i, if its write set is non-empty, writes a value that is different from X_i's initial value.

Transactions read and write sets			
T	T_1	T_2	T_3
R_T	$\{X_2\}$	$\{X_3\}$	$\{X_1\}$
W_T	$\{X_1\}$ if R(X_2)=⊥, else {}	$\{X_2\}$ if R(X_3)=⊥, else {}	$\{X_3\}$ if R(X_1)=⊥, else {}

Consider the following executions. For each $i = 1, 2, 3$, in any of the following executions, if it includes T_i then its coordinator handler is executed by C_i.

Solo Executions. Let E_1, E_2, E_3 be failure-free synchronous executions of I, where transaction T_i runs solo in E_i. Since E_i contains a single transaction and I satisfies weak progress, transaction T_i commits in E_i. Since E_i is synchronous, has no failures and contains only T_i, and I satisfies fast decision, T_i is fast deciding in E_i.

▷ **Claim.** T_i must have a depth of at most 4 in E_i.

To see this, note that by the definition of fast decision, if transaction T_i in E_i has depth at least 3, the empty prefix of E_i must have an extension C_i of partial depth $d_{E_i}(T_i, C_i) \le 2$ in which the value of the read set's item is known by some process to be decided, and therefore the write set is known by that process to be decided as well. By Corollary 3.7, the depth of T_i in E_i must be at most 2 more than the depth of C_i, and therefore is at most 4.

	X_1	X_2	X_3
N_1		1	2
N_2	2		1
N_3	1	2	

■ **Figure 2** Visual representation of execution E_{concur} in the proof of Theorem 4.2. The numbers in the table represent the order of writing on each node; on node N_1, X_2 is written first, followed by X_3, and so on.

Since I satisfies 1-seamless fault tolerance, there exist executions E_1', E_2', E_3' of I, where the first event in E_i' is a crash of N_i, T_i runs solo and the depth of T_i in E_i' is the same as its depth in E_i. We assume that in each E_i', a different set of processes runs the handlers. Lastly, since I is serializable, each E_i' is serializable, thus T_i's read in E_i' returns the initial value of $X_{(i \bmod 3)+1}$, and therefore modifies X_i as part of its write set.

Since T_i's write set is only determined from the outcome of T_i's read, and may be empty until that read's value is decided, by Lemma 4.3, no base object modifications visible to other transactions are executed by T_i in E_i' until after T_i's read set values are known to some process. Let the shortest prefix at which some process gains knowledge of T_i's read set values in E_i' be P_i. By Lemma 4.6, $d_{E_i'}(T_i, P_i) \geq 2$.

Concurrent Execution.   We define an execution E_{concur} with all 3 transactions. In E_{concur}, all messages between processes on node N_i and any process that executes handlers associated with T_i are arbitrarily delayed. For each $i = 1, 2, 3$, let processes that execute T_i in E_i' run in E_{concur} identically to E_i', in the same order of steps, until the end of P_i.

Note that up to this point, E_{concur} is indistinguishable to all executing processes from the solo executions, since none of them has made any shared memory modifications visible to the others. Therefore, the prefix $P_{knowledge}$ of E_{concur} up to this point is an execution of I.

We continue E_{concur} as follows: Let all messages sent on behalf of T_i at depth $d_{E_i'}(T_i, P_i)$ be sent in E_{concur}, and be received and handled in the following order: on node N_1, messages for T_2 are received first, and their handlers are run to completion, followed by messages for T_3. On node N_2, T_3's messages are handled first, followed by messages of T_1. Finally, on node N_3, messages of T_1 are handled first followed by messages of T_2. coordinator handlers receive messages in the same order they received them in their corresponding solo executions.

Recall that transaction T_i reads data item $X_{(i \bmod 3)+1}$ and, if it reads the initial value, writes data item X_i. Thus, the service order defined above for execution E_{concur} (see the order in which the nodes process their writes in Figure 2) means that on each node, the second serviced transaction writes to data item X after the first transaction reads X, but it is never the case that a transaction reads a data item after it was written by another transaction on the same node. Since the data set of the second transaction to execute handlers after prefix $P_{knowledge}$ on each node does not overlap with the write set of the first one, and since I provides invisible reads and DAP, then by Lemma 4.4, the first transaction does not make base object modifications visible to the second transaction. In other words, on each node, a process executing the second transaction cannot observe any changes on shared memory. Thus, E_{concur} is still indistinguishable from E_i' to any node process that executes T_i up to the end of the handlers of messages sent at depth $d_{E_{concur}}(T_i, P_i)$. Note, however, that since $d_{E_i'}(T_i) \leq 4$ and $d_{E_{concur}}(T_i, P_i) \geq 2$, this means that E_{concur} remains indistinguishable from E_i' to these processes until T_i is decided.

Therefore, for all three transactions T_i commit in E_{concur}, reading the initial value of $X_{(i \bmod 3)+1}$ and writing a non-initial value in X_i. However, this yields a circular dependency between the transactions (transaction T_1 must happen before T_2, which must happen before T_3, which must happen before T_1), which contradicts serializability.   ◄