Temporary Availability: A Practical Reframing of the CAP Trade-off in the Traquest Model

Dániel Balázs Rátai[1,*], Zoltán Horváth[1], Zoltán Porkoláb[1] and Melinda Tóth[1]

[1]Eötvös Loránd University, Faculty of Informatics, Pázmány Péter sétány 1/C, 1117 Budapest, Hungary

Abstract

The CAP theorem is one of the most widely cited results in distributed systems, yet in everyday engineering, it is often reduced to a binary slogan: a distributed system is either consistent or available. We argue that for the workloads that hurt the most in practice—long chains of *dependent*, nested operations—this framing hides the real question, which is not *whether* a consistency–availability trade-off exists, but *where* on the execution path it is paid. Building on the Traquest model, an application-layer ACID concurrency model in which a transaction is a tree of revocable, temporarily responding computation units (Traquests), we introduce and semi-formally characterise the notion of *temporary availability*. A temporary response (a Trasponse) is delivered within a bounded time, even under network partitioning, can be consumed by the rest of the computation, and remains revocable until the surrounding transaction tree commits. This relocates the CAP-induced synchronisation pressure from the critical path of dependent steps to a single commit phase, without violating the CAP theorem and while maintaining strong consistency over the transaction as a whole. We position temporary availability as a well-defined intermediate point between strong and eventual consistency, relate it to PACELC, the saga pattern and optimistic replication, provide an illustrative latency-cost model, and walk through the behaviour of the model under network partitioning using a spatial-index example. We discuss the validity conditions and limits of the approach, as well as its implications for the reliability and quality of distributed software. We illustrate conflict detection, undo and re-execution on a worked example, and we report preliminary measurements from a containerised prototype: on a read workload with genuine data dependencies, where request pipelining is not applicable, a Traquest-based prototype key–value store sustained roughly two orders of magnitude more operations per second than a widely used in-memory store under the same conditions, even in fully ACID mode.

Keywords

distributed systems, CAP theorem, temporary availability, ACID, transactional consistency, partition tolerance

CEUR-WS.org/Vol-4336/paper20.pdf

1. Introduction

The CAP theorem states that in the presence of network partitioning, a distributed data store cannot simultaneously guarantee strong consistency and full availability [1, 2, 3]. The result is foundational, but its everyday use is often reduced to a binary classification of systems as "CP" or "AP". This simplification obscures a question that is far more useful for an engineer: in a real workload, at which point of the execution path is the trade-off actually paid, and how painful is it there?

The cost is rarely uniform. In practice, the most damaging latency problems do not arise from isolated, single-step operations, but from *chains of dependent operations*: the output of one step is the input of the next, and the computation iterates across several components or parts of the global state. In a classical, database-centric multi-tier architecture, each such dependent step typically forces its own network round trip and its own consistency decision (a lock, a quorum, a validation). The latencies add up, and the caution induced by CAP—locking, quorum coordination—makes the accumulation worse. For short, independent operations, the relative impact of CAP is usually smaller because other costs (user interaction, more complex business logic, external services) dominate anyway.

SQAMIA 2026: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 2026, Novi Sad, Serbia

*Corresponding author.

danielratai@inf.elte.hu (D. B. Rátai); hz@inf.elte.hu (Z. Horváth); gsd@inf.elte.hu (Z. Porkoláb); toth_m@inf.elte.hu (M. Tóth)

0000-0002-9587-571X (D. B. Rátai); 0000-0001-9213-2681 (Z. Horváth); 0000-0001-6819-0224 (Z. Porkoláb); 0000-0001-6300-7945 (M. Tóth)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-19

This paper revisits the CAP trade-off from the perspective of the Traquest model [4, 5], an application-layer concurrency model that enforces ACID properties without delegating coordination to a separate storage tier. In the Traquest model, a transaction is not a sequence of independent "small transactions" but a single hierarchical *Traquest tree*; the tree either commits as a whole or the affected parts are rolled back and, where appropriate, re-executed. Within this context, we make the following contributions.

1. We reframe the CAP trade-off for dependent computations: the trade-off does not disappear, but it is *relocated* from the critical path of the dependent steps to the commit phase of the transaction (Section 3).
2. We introduce and semi-formally define *temporary availability*, a well-defined intermediate point between strong and eventual consistency, in which every dependent step receives a bounded-time, usable, but revocable response (Section 4).
3. We characterise the behaviour of the model when partition tolerance is required, including conflict reconciliation and the consistency a client observes during the temporary phase (Section 5), and we walk through conflict detection, undo and re-execution on a concrete example, first informally and then at the level of individual messages (Section 5.4).
4. We give an illustrative latency cost model and a worked spatial-index scenario that make the relocation of the trade-off quantitatively concrete (Sections 6 and 7).
5. We report preliminary measurements from a containerised prototype benchmark on a dependency-bound read workload (Section 8).
6. We discuss the validity conditions, limits, and software-quality implications of the approach (Section 9).

2. Background and Related Work

2.1. The CAP theorem and PACELC

In its classical form, *consistency* typically means that a read returns the most recent write (or an error), *availability* means that every request receives a non-error response within bounded time (not necessarily the most recent value), and *partition tolerance* means that the system keeps operating even when messages between nodes are delayed or lost [1, 2]. The essential statement is that under a partition, the designer must choose between consistency and availability, because the two cannot be guaranteed simultaneously. As Gilbert and Lynch emphasise [3], this is a statement about an unavoidable limit, not a critique of a particular architecture; many different design strategies can live inside that limit.

PACELC [6] refines CAP by observing that even in the absence of a partition (the "else" case) a system still trades latency (L) against consistency (C); only under a partition (P) does it trade availability (A) against consistency (C). This refinement is directly relevant to dependent operation chains, because their dominant cost in the partition-free case is precisely the accumulated latency of synchronising each step.

2.2. Consistency models

Strong consistency models provide robust correctness guarantees, but their applicability is limited in large-scale, geographically distributed systems because they pay for those guarantees with higher latency and reduced availability [3]. Eventual consistency [7] takes the opposite stance: it always responds immediately, but the response may be stale, and the burden of handling transient inconsistencies is pushed onto the application. Compromise solutions often rest on implicit assumptions about the temporal behaviour of the system—for example, that inconsistent states are short-lived, or that certain conflicts are rare—and these assumptions do not always hold in a distributed setting.

2.3. Recovery without global locking: sagas and optimistic approaches

Several established techniques avoid global locking by allowing temporary inconsistency and resolving it afterwards. The saga pattern [8, 9] decomposes a long-running business transaction into a sequence of local steps, each with an explicit *compensating* action, so that atomicity-like recovery is achievable without distributed locks. Optimistic replication [10] and optimistic concurrency control [11] permit reads and writes without prior coordination, detect conflicts after the fact, and resolve them by reconciliation. These approaches all "bet" on a low conflict rate and defer coordination—a stance the present work shares—but they differ in where the recovery logic lives, as we discuss in Section 9.1.

2.4. The Traquest model in brief

We recall only the elements of the Traquest model needed for this paper; full treatments are given in [4, 5]. The name *Traquest* combines *transaction* and *request*. A Traquest is the atomic unit of computation: one can send a request to a Traquest, and it answers, but the answer establishes a permanent parent–child binding for the duration of the surrounding transaction. Because of this, a Traquest can be rolled back even after it has answered.

Three properties matter here.

Traquest tree. Traquests call other Traquests, forming a tree. A transaction is exactly one such tree. If any branch fails, the failure cascades up to the first Traquest that has not yet responded; that Traquest rolls back itself and its descendants and re-executes. When all branches have responded, the root initiates a commit. This yields atomicity and isolation for the tree as a whole.

Trasponse (temporary response). When a Traquest receives a request, it can immediately perform local reads and writes and reply at once with a *temporary* response, the *Trasponse*, while the effects are still being synchronised. A Trasponse is, in effect, a response that says: "Here is my immediate, very probably correct answer; I may still have to confirm it, and I will let you know if it must be undone."

Data Protector. A Data Protector manages a given record of the global state and protects it from conflicting reads and writes. It uses the hierarchical timestamps [12] of the calling Traquests to detect conflicts; on a conflict, it triggers the undo of an already-responded Traquest and serves the correct order. Data Protectors are the only entities that touch durable storage, so the physical storage is abstracted away.

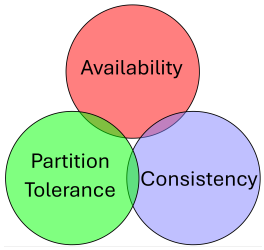
Crucially, durability and multi-node consistency are achieved by adding Traquest-tree branches that replicate write operations; these synchronisation messages can be buffered and committed together, so the "hard" coordination is concentrated at the boundaries of the tree rather than spread across every dependent step. This concentration is the structural property that the present paper exploits.

3. Reframing the CAP Trade-off in the Traquest Tree

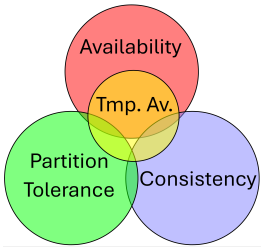
The Traquest-tree viewpoint does not bypass the CAP problem; it reorganises it. The right practical question is not whether a trade-off exists, but *where* on the execution path it appears.

We stress at the outset that the central claim is *not* about user-interface responsiveness. A Trasponse may occasionally surface in a UI and make an application feel more responsive, but that is a side effect. The scientific claim is structural: the model rearranges the execution of dependent operation chains so that the steps inside a tree are not forced into a global, commit-like synchronisation every time; the atomic semantics applies to the tree as a whole.

Inside a Traquest tree, the practically most expensive CAP-induced dilemmas—the ones on the chained, dependent steps—are removed. Dependent steps can proceed without an immediate, global validation (quorum/lock/coordination) at each step; if a conflict arises, it is resolved within the tree context through undo and, if necessary, re-execution. Two direct consequences follow.



(a) Original



(b) With the Traquest model

Figure 1: The CAP Venn diagram. Under the Traquest model, the region of availability is extended by a band of *temporary availability* ("Tmp. Av."): on the critical path of dependent steps, the system stays responsive with temporary results, while the strict consistency–availability trade-off is concentrated at the global commit.

1. **Dependent-step latency does not accumulate because of commit pressure.** In classical request–response architectures, dependent steps are realised as a chain of "small transactions", each requiring its own network round trip and consistency decision. Within a Traquest tree, steps may instead advance with temporary results, while possible conflicts are handled later in the tree context. For long, chained processes, the total latency can therefore drop substantially.
2. **The "hard" synchronisation concentrates at the end of the tree.** The cost of final commitment appears chiefly in the commit phase, where the state of the tree must be closed in a way that satisfies durability and replication. In other words, the CAP tension appears not "at every step" but predominantly at the close. This is favourable because, in most systems, it is exactly the many dependent steps that accumulate latency, and the Traquest-tree context essentially removes that.

From the CAP viewpoint, the temporary response complements the notion of availability. The classical definition requires that every request receive a bounded-time response. However, in practice, many operations are dependent (built on the result of previous steps), so a single network uncertainty can stall an entire chain. Within a Traquest tree, the system allows the steps to advance with temporary results and to handle conflicts later with targeted undo/re-execute. Figure 1 illustrates that, under the Traquest model, the domain of "availability" is extended to the internal context of the transaction tree: the CAP trade-off still exists at the global commit, but not on the critical path of the dependent steps.

Put differently, the Traquest tree effectively establishes a context in which arbitrarily complex, dependent computations can be carried out as though the constraints of the CAP theorem did not apply to them: inside the tree, every step stays available and consistent, and the CAP trade-off surfaces only once, at the very end, when the tree commits.

4. Temporary Availability

4.1. An intermediate notion

The key concept of this paper is *temporary availability*. It is worth delimiting it precisely against the two extremes. A classical, strongly consistent system may block the service of a dependent operation until global coordination completes; when partition tolerance is required, preserving consistency may then force a trade-off with availability. Eventual consistency, by contrast, always answers immediately, but the correctness of the answer is not guaranteed, and the application must explicitly handle transient inconsistencies.

Temporary availability occupies a well-defined intermediate position. The system *always answers* a dependent step—with a Trasponse—so it is available in the operational sense; yet this answer is, in the semantics of the model, explicitly *temporary*: it is revocable until the surrounding Traquest tree commits. The difference from eventual consistency is therefore not whether the state "may be temporarily incorrect", but *who handles it and how*: in the Traquest model, the temporality and its resolution (undo + re-execute) are carried by the model itself, not by the application developer. The

guarantee of strong consistency is not lost: it applies to the Traquest tree as a whole, at the moment of commit.

4.2. A semi-formal definition

We now make the relationship to CAP precise. Recall the standard notions: a system is *available* if it returns a non-error response within a bounded time to every request it receives, and *partition tolerant* if it keeps operating when messages between its nodes are delayed or lost [1, 2].

Let $r(t)$ denote the response to a request at time t. In the classical formulation, $r(t)$ is either *final* (and then it must be consistent) or an *error*. The Traquest model introduces a third possibility: $r(t)$ may be a *temporary* (Trasponse) response that

(i) can be produced within bounded time *even under network partitioning*;
(ii) is *usable* in the computation branch that consumes it; and
(iii) is *revocable*: if, before the commit, a conflict at a corresponding durable state is discovered, the subtree built on it can be repaired by undo and re-execution.

Finality is thus bound not to the act of responding but to the commit. With this move, *responsiveness* (there is always a bounded-time, possibly temporary response) and *eventual strong consistency* (restored at commit) can be separated.

4.3. Relationship to PACELC

In PACELC terms [6], the Traquest approach can be described as a system that, in the partitioned case (P), protects consistency by deferring the commit (while remaining available to dependent steps), and, in the partition-free case (E), optimises latency (L) while preserving eventual strong consistency (C). The most significant gain is on the "else" branch: even in the partition-free case, the bottleneck of dependent, chained steps is their accumulated latency, and the model reduces it with temporary responses while concentrating consistency at the commit. This contradicts neither CAP nor PACELC; it shows *where* the trade-off is least painful—moved from the critical path of many dependent steps to the single close of the transaction.

5. Behaviour When Partition Tolerance Is Required

5.1. Separating progress from commitment

The CAP theorem is specifically about network partitioning, so it is worth examining how the model behaves when partition tolerance is required. The approach separates the *progress* of the computation from its *final commitment*. A Traquest tree can continue executing its dependent steps with temporary responses regardless of current network conditions, because these responses are interpretable within the tree and rely on locally reachable Data Protectors. The dependent steps, therefore, never block: availability on the critical path is preserved.

What may be delayed is the *commitment* of the tree. A tree's commit needs to wait only when the tree depends on another Traquest tree that has not yet committed; in that case, it waits for the other tree to commit. Such a dependency is never mutual: the hierarchical timestamps [12] impose a strict order on the transactions, so two trees cannot wait on each other, and no deadlock can arise. The "hard" coordination is thus not only concentrated at commit (Section 3); it is also limited to genuine cross-transaction dependencies, rather than being paid at every dependent step.

Replication for fault tolerance fits the same picture. A replica always lives on a different node—this is precisely what makes it a replica and what allows the data to be recovered if a node is lost—and a write is replicated by adding tree branches that repeat it on the other nodes [4]. These branches are part of the same transaction and are committed together. If network partitioning makes a replica temporarily unreachable, the dependent steps still proceed with temporary responses; only the synchronisation of the affected branch is deferred until connectivity returns.

5.2. Reconciliation after temporary inconsistency

If temporary results turn out to be inconsistent—for example, because synchronisation across nodes later reveals a conflicting order—the model applies its usual conflict-resolution mechanism: the Data Protectors roll back the affected (sub)tree and re-execute it with a fresh, later timestamp. In this way most of the temporary work can be preserved, and only the inconsistent parts are resolved, in a targeted manner, without discarding the whole transaction.

5.3. Client-observed consistency during the temporary phase

It is essential to clarify what a client sees while responses are temporary. Within a single Traquest tree, the reads are mutually consistent: the tree semantics guarantee that the steps of the same transaction observe a unified, isolated view of the affected state. The temporary values, however, are not published as final to other transactions; another tree sees only committed values as final. After commitment (or, if needed, undo and re-execution), the application invariants hold, because the commit conditions are exactly what enforce them, and the externally observable end result is atomic: as if the transaction had run once, consistently.

As a limitation, if a client *acts* on a temporary response that is later revoked, the application boundary must be prepared for it (compensation), similar to the pattern of optimistic user interfaces [13]. In practice, this is rarely a burden: in the vast majority of cases, one need not deal with the temporary responses outside the tree at all, because it is enough to await the single final response, exactly as in a traditional request–response interaction. In other words, with respect to what the application boundary must handle, the Traquest model leaves the developer essentially where the classical request–response model does; the gain is in latency, since the round-trip times of the dependent steps no longer accumulate. The Traquest model also localises the remaining question to the boundary of the tree: inside the tree, the rollback is automatic, while handling already-externalised effects outside the tree remains an application decision.

5.4. A worked example of conflict detection and resolution

The mechanism described in the preceding subsections is easier to follow on a concrete execution than in prose, so we walk through it twice: first informally, at the level of the Traquest tree (Figure 2), and then at the level of individual messages (Figures 3 and 4). The two views differ in the scope of the conflict—within a single transaction in the first case, between two independent transactions in the second—but they show the same story: dependent steps receive Trasponses without waiting for global coordination, a conflict is discovered afterwards, the affected subtree is undone and re-executed, and the surrounding tree finally commits as a whole.

Figure 2 follows a single tree through six stages. In stage 1, the root $T1$ issues requests to its children; $T1/1$ and $T1/3/1$ read from Data Protector $DP1$ and answer immediately with Trasponses, so the branches that depend on them can continue at once. In stage 2, $T1/2$ writes to the same record of $DP1$. The Data Protector compares the hierarchical timestamps of the operations it has served and finds that the write is logically ordered before the read that $T1/3/1$ has already consumed: the temporary answer is no longer valid, and a conflict is signalled. In stage 3, $DP1$ initiates the undo of $T1/3/1$; because $T1/3$ has not yet responded, the rollback cascades over its subtree, and $T1/3/2$ is rolled back with it. In stage 4, the affected subtree is re-executed from $T1/3$; note that the work of $T1/1$ and $T1/2$ is untouched—only the part built on the invalidated value is repeated. In stage 5, the re-executed $T1/3/1$ reads $DP1$ again, this time in an order the Data Protector accepts. In stage 6, the remaining branch writes to $DP2$, every branch has responded, and the root commits the tree. The important observation is that no step in this execution ever waited for a global validation: the coordination happened once, at the close of the tree, and the repair was local to the invalidated subtree.

The same mechanism can be examined message by message, this time across transaction boundaries. Figures 3 and 4, adapted from [4], show two transactions, $T1$ and $T2$, incrementing the same record through the Data Protector DP_i, which is the only entity that touches $Storage_i$. In part (a), $T1/1$ reads

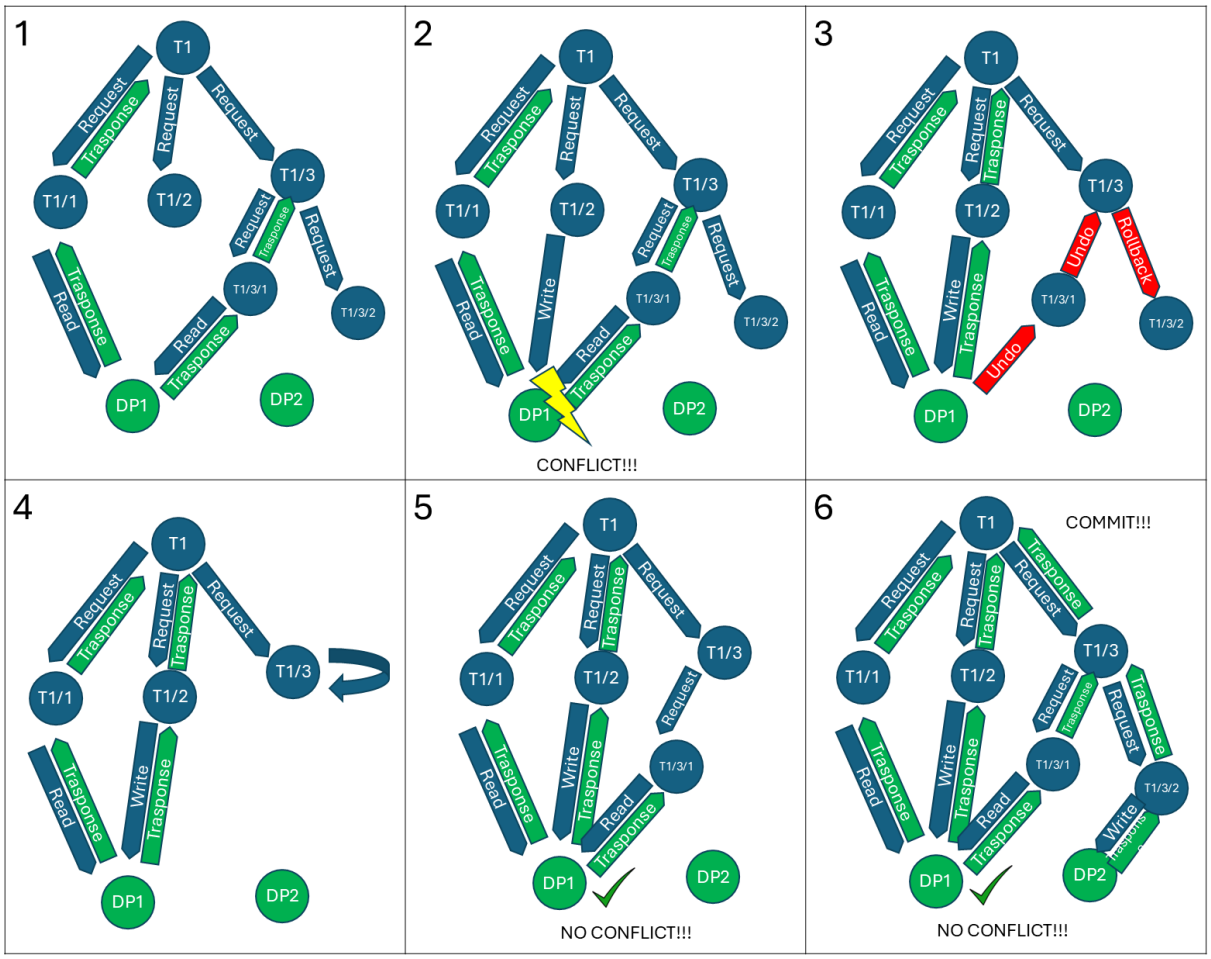
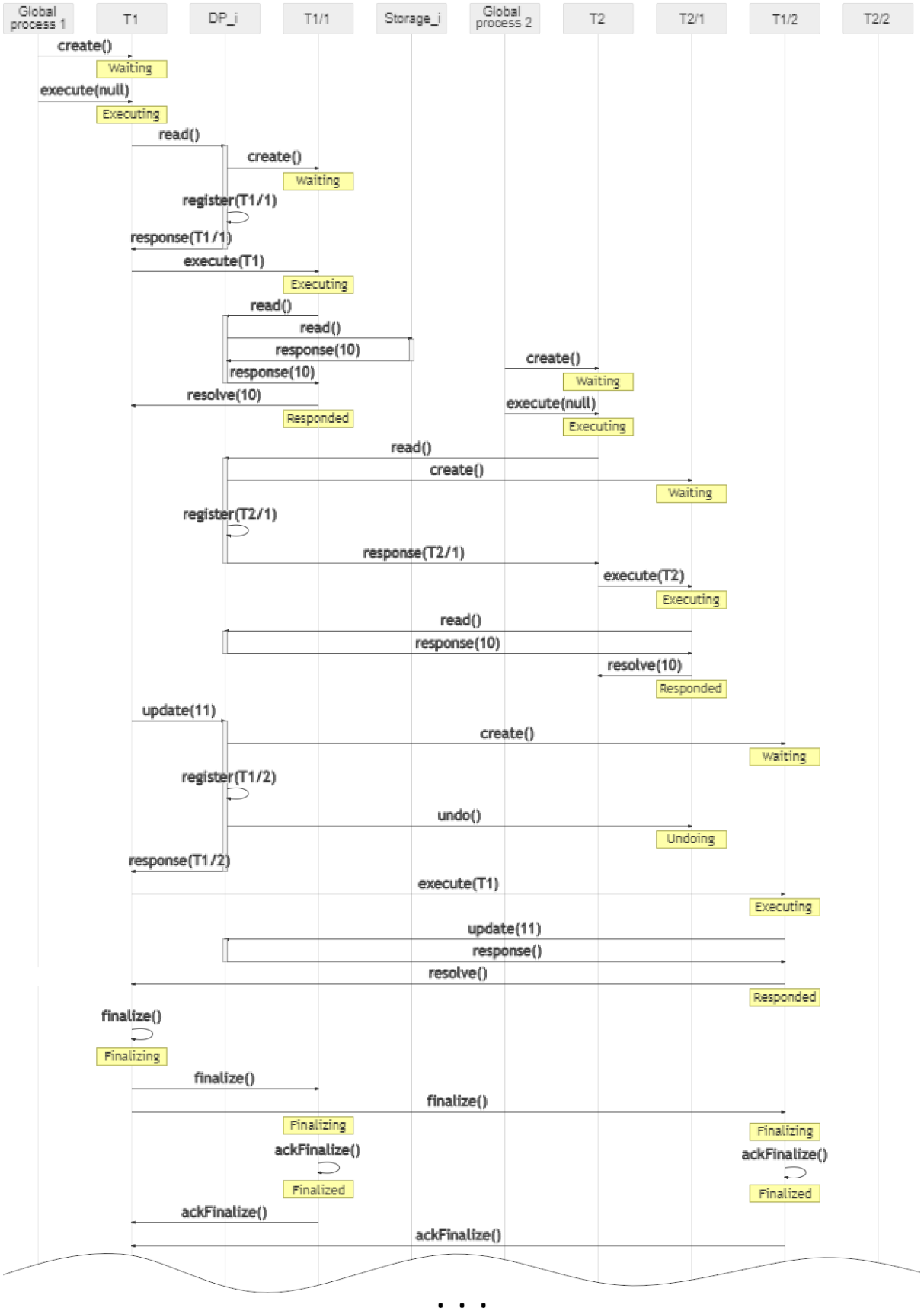


Figure 2: Conflict detection and resolution inside a Traquest tree. The conflict is discovered while the tree is still executing; a targeted undo and re-execution of the affected subtree follows, and the tree eventually commits without conflicts. Requests, reads and writes are drawn in dark blue, temporary responses (Trasponses) in green, and undo and rollback messages in red. $DP1$ and $DP2$ are Data Protectors.

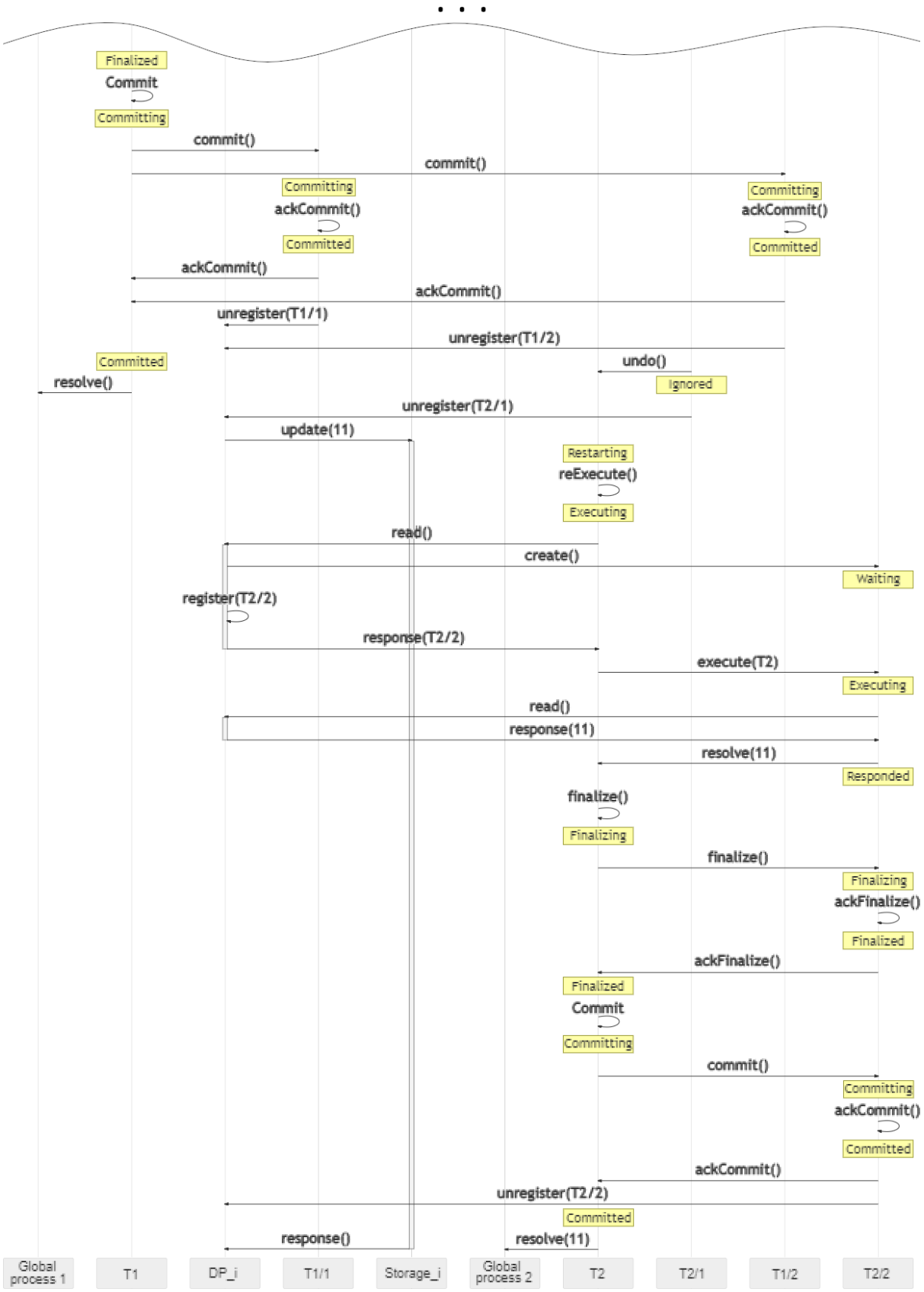
the record and resolves with the value 10; the enclosing Traquest enters the *Responded* state, which is exactly the state in which a Trasponse has been delivered but nothing has been committed. $T2$ then reads the same record and also receives 10, and it too responds. When $T1$ issues its `update(11)`, DP_i registers the write, recognises that the read already served to $T2/1$ can no longer be part of a consistent order, and sends `undo()` to it—while $T1$ continues undisturbed towards `finalize()`. In part (b), $T1$ commits and its operations are unregistered; the undo of $T2/1$ completes (its result is *Ignored*), $T2$ enters *Restarting* and calls `reExecute()`. The repeated read now returns 11, and $T2$ finalises and commits on top of the value $T1$ actually wrote. Neither global process ever blocked, no lock was held across the dependent steps, and the externally visible outcome is the same as a serial execution of the two transactions.

Two remarks are in order about how this example relates to network partitioning. First, neither figure is drawn as a partition scenario: in Figure 2 the conflict arises between sibling subtrees of a single transaction, and in Figures 3 and 4 between two independent transactions. This is not a limitation of the example, because nothing in either diagram assumes that the participants share a machine. Traquests and Data Protectors communicate only by asynchronous messages, and the protocol is indifferent to where each of them runs; placing a subtree or a Data Protector on another node changes the latency of the arrows, not their sequence. A partition is then simply the case in which some of those messages are delayed for a long time: the dependent steps continue on Trasponses exactly as shown, and when connectivity returns and the deferred synchronisation reveals a conflicting order,



(a) Trasponses are delivered, the conflict is detected, and `undo()` is sent to the affected Traquest.

Figure 3: Message-level view of the same mechanism: two transactions incrementing one record through a Data Protector, with conflict detection, undo and re-execution (adapted from [4]). Part (a) continues in Figure 4.



(b) The first tree commits; the affected Traquest is re-executed and commits on top of the corrected value.

Figure 4: Continuation of Figure 3.

Table 1
Illustrative latency comparison at a fixed total number of steps ($n_{\text{all}} = 1000$, $C_{\text{local}} = 10$ ms; assumed parameters). Only the number of dependent steps changes; the Traquest side is approximately constant.

Dependent steps (n_{dep})	$L_{\text{classical}}$	L_{Traquest}
10	17 ms	11 ms
100	80 ms	11 ms
1000	710 ms	11 ms

what follows is precisely the sequence in the figures—the Data Protector signals the conflict, the affected subtree is undone, it is re-executed with a fresh timestamp, and the tree commits as a whole. What a partition changes is when a conflict becomes visible, not how it is resolved. A node-level diagram would additionally have to show the layer that carries and buffers inter-node traffic, which is a matter of implementation rather than of the model, and we omit it here. Second, the figures make the cost structure of Section 6 tangible: the messages that must be exchanged in bulk are those of the finalize and commit phases at the bottom of the diagrams, whereas the dependent reads in the upper half are answered immediately.

6. Illustrative Cost Model

To provide quantitative intuition for the relocation of the trade-off, consider the end-to-end latency of a chain of dependent operations. Let n_{dep} be the number of dependent (chained) steps, n_{all} the total number of steps, RTT a typical network round-trip time, C_{coord} the per-step coordination cost (lock/quorum/validation), C_{step} the local computation cost of a single step, $C_{\text{local}} = n_{\text{all}} \cdot C_{\text{step}}$ the total local computation, and C_{commit} the overhead of final commitment (durability + replication). Then, approximately,

$$L_{\text{classical}} \approx n_{\text{dep}} \cdot (RTT + C_{\text{coord}}) + C_{\text{local}}, \tag{1}$$

$$L_{\text{Traquest}} \approx C_{\text{commit}} + C_{\text{local}}. \tag{2}$$

The C_{local} term arises in both approaches. If the total number of steps is fixed and only the fraction of dependent steps varies, the Traquest side stays approximately constant; if every step is dependent ($n_{\text{all}} = n_{\text{dep}}$), the C_{local} term grows on both sides. Because C_{step} is typically very small relative to RTT, C_{coord} and C_{commit}, the Traquest side tends towards $O(1)$ while the classical side tends towards $O(n_{\text{dep}})$; for large C_{step} (computation-heavy steps) the gap between the two approaches narrows.

The following figures are *assumed, illustrative* values (not measurements); their purpose is to show the order of magnitude. Assume a local (LAN) cluster with $RTT = 0.5$ ms, $C_{\text{coord}} = 0.2$ ms, $C_{\text{commit}} \approx 1$ ms and $C_{\text{step}} \approx 0.01$ ms.

(1) Fixed total number of steps. If the total work is constant ($n_{\text{all}} = 1000$, so $C_{\text{local}} = 10$ ms) and only the fraction of dependent steps changes, the Traquest-based latency stays essentially constant while the classical one grows linearly (Table 1).

(2) All steps dependent. If every step is dependent ($n_{\text{all}} = n_{\text{dep}} = n$, so $C_{\text{local}} = n \cdot C_{\text{step}}$), the local computation grows on both sides, but the classical side is dominated by the $n \cdot (RTT + C_{\text{coord}})$ term. The difference remains of a different order of magnitude (Table 2).

The shared lesson of the two cases is the essence of the reframing: as long as the per-step computation C_{step} is small relative to RTT, C_{coord} and C_{commit}, the Traquest-based execution stays close to $O(1)$, while the classical one grows as $O(n_{\text{dep}})$. If the local computation cost becomes large, the C_{local} term dominates on both sides, and the gap narrows. In the worst case—many conflicts with repeated undo/re-execute—the C_{local} term can grow further, an effect we return to in Section 9.

The figures above are analytical: they follow from the cost model, not from a measurement. Section 8 reports a preliminary benchmark on a prototype implementation that was run under conditions matching

Table 2
Illustrative latency comparison when all steps are dependent ($n_{\text{all}} = n_{\text{dep}} = n$; assumed parameters). The Traquest side stays near C_{commit} while the local computation is small.

Steps (n)	$L_{\text{classical}}$	L_{Traquest}
10	7.1 ms	1.1 ms
100	71 ms	2 ms
1000	710 ms	11 ms

Table 3
Inserting a point into an octree: a database-centric multi-tier architecture versus the Traquest model.

Aspect	**Multi-tier (DB-centric)**	**Traquest model**
Messages for the example insert	about 44 [4]	about 6
Coordination per dependent step	a lock/quorum decision at each step	none on the critical path; temporary responses
When partition tolerance is required	dependent reads block to stay consistent, so availability is sacrificed	dependent steps proceed with temporary, revocable responses
Consistency guarantee	per operation, at the database	atomic over the whole tree, at commit
Conflict handling	locking, or transaction abort	undo, and re-execution of the affected subtree

the assumptions of this model—a read workload with genuine data dependencies, where the operations cannot be pipelined—and the measured difference is of the order the model predicts.

7. Worked Scenario: Point Insertion in a Spatial Index

We illustrate the behaviour on an octree-based [14] spatial index, the kind used in geographic information systems. Inserting a point requires several dependent reads and writes (from the root down to a leaf and then a recursive refresh of the parents). In a multi-tier, database-centric architecture, this dependency implies many network round-trips (about 44 messages in the analysis of [4]), because each step must be read, checked and locked on the data servers before the next one can proceed. A Traquest tree typically resolves the same task in a handful of messages (about 6): the dependent reads return temporary responses and need no per-step synchronisation, and the writes are aggregated and synchronised once, at the close of the tree. Table 3 contrasts the two architectures on this task, including their behaviour when partition tolerance is required.

The contrast is instructive. A strictly consistent multi-tier system keeps every dependent step on the critical path of coordination, and to stay consistent when partition tolerance is required, it must block a dependent read rather than risk returning a stale value—so it sacrifices availability. An eventually consistent store remains available but provides no atomic rollback semantics for the nested computation. The Traquest model sits between the two: it remains available on the critical path via temporary responses, and it still secures atomicity by deferring it—with revocability—to the commit.

8. Preliminary Experimental Evaluation

The cost model of Section 6 predicts that the advantage of the Traquest model grows with the number of dependent steps, because those steps no longer pay a network round trip and a coordination decision each. To see whether this effect is observable in practice, we built a containerised benchmark around a prototype implementation. We report it as preliminary evidence rather than as a full evaluation: the prototype is incomplete, and the measurement covers conflict-free reads only.

Table 4
Read throughput of Redis and of the TraquestDB prototype in a containerised benchmark on the same host. The last two columns give the speed-up of the two TraquestDB modes over the corresponding Redis configuration. Conflict-free reads; the prototype is incomplete and writes were not part of this measurement.

System and configuration	Reads/s	TraquestDB speed-up	
		ACID	Eventual
Redis, no pipelining	81 234	5.7×	21.9×
Redis, pipelining	508 388	0.9×	3.5×
Redis, full round trip	5 376	86×	331×
TraquestDB, full ACID	464 684	—	—
TraquestDB, eventual consistency	1 779 359	—	—

8.1. Setup

We compared TraquestDB, a prototype key–value store built on the Traquest model, with Redis, one of the fastest in-memory stores in wide use. Both systems ran in separate Docker containers on the same host, an Intel Core i7-9700 at 3.00 GHz with 32 GB of memory, running 64-bit Windows 10 Pro. The load was generated by a separate `node.js` application on the host, which sent read operations to the containerised service over a TCP connection and measured the responses. Running both systems on identical hardware, in an identical containerised operating environment, and driving them with the same load generator removes the most obvious sources of difference between the two configurations.

One asymmetry is worth stating at the outset, because it works against the Traquest side: the TraquestDB prototype runs in a TypeScript/Node.js environment, whereas Redis is a native system written in C. The difference reported below therefore does not come from a faster language or runtime; it appears in spite of the runtime handicap.

8.2. Workload

The workload was designed to be representative of the situation the model targets: a transaction that performs many operations built on each other's results. In that situation, pipelining is not applicable, because a request cannot be issued before the previous response has arrived. We therefore measured Redis in three configurations. The first two follow the parameterisation of the Redis documentation [15, 16], which is designed to show the maximum throughput of the server: without pipelining, and with pipelining enabled. These served as a sanity check on our containerised environment—the documentation reports roughly 72 145 operations per second for the comparable 50-client read case, and we measured 81 234, so our environment is not anomalous. The third configuration is a *full round trip* test, in which the load generator waits for each response before sending the next operation. This is the configuration that corresponds to a dependent chain, and it is the one that should be compared with the Traquest side.

TraquestDB was measured on conflict-free reads in two modes: fully ACID, and with eventual consistency, which the prototype also supports for reads.

8.3. Results

Table 4 summarises the measurements. On the comparison that matches the workload of interest—Redis in full-round-trip mode against TraquestDB—the prototype performed roughly $331\times$ more read operations per second in eventual-consistency mode and roughly $86\times$ more even in fully ACID mode. Against Redis without pipelining the factors are about $22\times$ and $5.7\times$; against the raw pipelined throughput of Redis, which no dependent workload can actually achieve, the eventual-consistency mode is still about $3.5\times$ faster, while the ACID mode is roughly on par.

The interesting comparison is not the largest number but the ACID one. A system that provides full ACID guarantees is being compared here with an eventual-consistency store, and it is nevertheless faster on a dependency-bound workload by a factor of roughly 86. This is what the reframing of Section 3 predicts: the guarantees are not paid for step by step, so the dependent steps do not accumulate round trips, and the price is deferred to a single commit.

8.4. Threats to validity

Several limitations must be kept in mind when reading these numbers.

- The measurement covers conflict-free reads. Under conflicts, rollbacks, or synchronisation across several nodes, the throughput of the Traquest side will drop; how far it drops depends on the conflict rate and the network conditions, as discussed in Section 9.
- Writes were not part of this measurement, and the prototype is not a finished system. The purpose was to estimate orders of magnitude in the workload region where the model is expected to help, not to publish absolute performance figures.
- The rows of Table 4 are not all like-for-like. The pipelined Redis row in particular measures a mode of operation that a dependent chain cannot use at all; it is included for completeness, and the full-round-trip row is the one that corresponds to the workload under discussion.
- The comparison is with the fastest kind of in-memory eventual-consistency store. Against an ACID SQL database—a comparison with matching guarantees—the difference would be expected to be larger, but that measurement was outside the scope of this work.
- The benchmark was run on a single host with containerised services, so the network round trips involved are local. In a geographically distributed deployment the RTT term of Section 6 grows on both sides, which would increase rather than decrease the relative advantage of the Traquest side.

9. Discussion

9.1. Positioning against related approaches

Sagas and compensating transactions. A saga [8] splits a long-running transaction into local steps, each with an explicit compensating action, achieving recovery without global locking. The difference is that in a saga, the developer must write the compensation step by step and reason about intermediate visibility for each step, whereas the Traquest model derives undo and re-execution generally from the tree semantics, and specifically targets the nested, dependent steps with temporary responses on the critical path. The similarity is that both avoid global locking and allow temporary inconsistencies that are resolved later.

Optimistic replication and optimistic concurrency control. These approaches [10, 11] allow reads and writes without prior coordination, detect conflicts after the fact, and resolve them by reconciliation. The difference is one of focus: optimistic replication typically concentrates on the convergence of *data*, while the Traquest model concentrates on the *computation* (the nested process), with revocable responses and tree-level atomicity, and at the processing layer rather than merely the storage layer [4]. The similarity is that both bet on a low conflict rate and defer coordination.

In summary, the distinguishing feature of the Traquest model is that it unifies the advantages of the above approaches—freedom from locking, deferred coordination—at the level of the nested, dependent transactional *computation*, with a general revocability that the developer need not code by hand.

9.2. Limits and validity conditions

For completeness, it is important to mark out *when* temporary availability brings a real gain and when it does not. The benefit appears when the workload is characterised by many dependent (chained) steps

and a low conflict rate: then the temporary responses can advance along the critical path and the "hard" synchronisation concentrates on a single commit. The illustrative computation in Section 6 reflects exactly this typical, optimistic case.

By contrast, at a high conflict rate, the C_{local} term—the cost of repeated undo and re-execution—can grow, and in the extreme can approach the classical, n-fold coordination cost. In that case, the benefit of temporary availability is diminished, because part of the tree's internal work must be redone. The model is therefore not a general "accelerator" but an optimisation for a particular workload profile (dependent steps, rare conflicts).

It is equally essential to be precise about when the final commit can be delayed. The dependent steps never block—they always proceed with temporary responses—but a tree's commit must wait when the tree depends on another Traquest tree that has not yet committed; it then awaits that tree's commit. Because the hierarchical timestamps impose a strict order on the transactions, such dependencies are never mutual, so two transactions cannot deadlock waiting on each other. When partition tolerance is required, the same separation holds: progress continues with temporary responses, while the durable, replicated result of a branch whose replica is temporarily unreachable is deferred until connectivity returns. This is consistent with the CAP theorem: the Traquest model changes the practical location of the trade-off, but does not eliminate the trade-off itself.

9.3. Implications for software quality

The reframing has a direct bearing on the quality of distributed software. Sagas, compensating patterns, and optimistic replication mark out the practical directions for responding to partitions, but they share a common trait: consistency must be restored by the programmer in explicit, coded form— whether in a saga with step-by-step compensating actions or in optimistic replication with subsequent reconciliation. By contrast, the Traquest model offers a unified semantic foundation: the atomic, revocable and re-executable semantics of the transaction tree makes the restoration of consistency a first-class part of the programming model, so that, at the programming level, the developer no longer has to deal with the question of compensation and intermediate visibility. Removing this hand-written, error-prone recovery logic is, in itself, a contribution to the reliability and maintainability of distributed systems.

10. Conclusion and Future Work

We have argued that for dependent, nested distributed computations the practically relevant question about the CAP theorem is not *whether* a consistency–availability trade-off exists, but *where* on the execution path it is paid. Building on the Traquest model, we introduced *temporary availability*: a well-defined intermediate point between strong and eventual consistency, in which every dependent step receives a bounded-time, usable but revocable response (a Trasponse), and strong consistency is restored across the entire transaction tree at commit. This relocates the CAP-induced synchronisation pressure from the critical path of the dependent steps to a single commit phase, without violating the CAP theorem. We gave a semi-formal characterisation, an illustrative latency cost model, a worked spatial-index scenario, a step-by-step and a message-level walkthrough of conflict detection, undo and re-execution, a preliminary containerised benchmark of a prototype implementation, a positioning against sagas and optimistic approaches, and an analysis of the validity conditions and limits.

This is a work-in-progress contribution. The most important open direction is the full formalisation of the temporary-availability semantics and of the underlying Traquest invariants—for example, in a machine-checked proof assistant—so that the correctness of conflict resolution when partition tolerance is required can be established with certainty. A second direction is to extend the preliminary evaluation of Section 8 beyond conflict-free reads: to writes, to controlled partitioning, and to a systematic sweep of the conflict rate, which is the parameter that determines where the benefit of temporary availability ends. A third direction is a finer, formal classification of the operation orderings that actually cause conflicts, which would let the conflict-resolution mechanism be optimised for specific workloads.

Acknowledgments

The authors thank their colleagues in the Department of Programming Languages and Compilers, Eötvös Loránd University, for their feedback, which helped clarify the Traquest model over several iterations.

Declaration on Generative AI

During the preparation of this work, the authors used Claude (Anthropic) in order to:

- Improve writing style;
- Grammar and spelling check;
- Citation management;
- Formatting assistance.

The scientific content of this paper is the authors' own work. The Traquest model, the notion of temporary availability, the cost model, the prototype implementation and every reported measurement were produced by the authors; earlier stages of this line of research have been published in [4, 5]. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] E. A. Brewer, Towards robust distributed systems (keynote), Keynote talk, PODC 2000, 2000. Principles of Distributed Computing (PODC '00).

[2] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, in: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC '02), 2002, pp. 51–59. doi:10.1145/564585.564601.

[3] S. Gilbert, N. Lynch, Perspectives on the CAP theorem, Computer 45 (2012) 30–36. doi:10.1109/MC.2011.389.

[4] D. B. Rátai, Z. Horváth, Z. Porkoláb, M. Tóth, Traquest model: A novel model for ACID concurrent computations, Acta Cybernetica 25 (2021) 435–468. doi:10.14232/actacyb.288765.

[5] D. B. Rátai, Z. Horváth, Z. Porkoláb, M. Tóth, Traquest model 2.0 – a model for faster ACID distributed computing, in: Distributed Computing and Artificial Intelligence, Special Sessions II, volume 1151 of *Lecture Notes in Networks and Systems*, Springer, 2025, pp. 255–260. doi:10.1007/978-3-031-80946-0_24.

[6] D. J. Abadi, Consistency tradeoffs in modern distributed database system design: CAP is only part of the story, Computer 45 (2012) 37–42. doi:10.1109/MC.2012.33.

[7] W. Vogels, Eventually consistent, Communications of the ACM 52 (2009) 40–44. doi:10.1145/1435417.1435432.

[8] H. Garcia-Molina, K. Salem, Sagas, in: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87), 1987, pp. 249–259. doi:10.1145/38713.38742.

[9] C. Richardson, Microservices Patterns: With Examples in Java, Manning, 2018.

[10] Y. Saito, M. Shapiro, Optimistic replication, ACM Computing Surveys 37 (2005) 42–81. doi:10.1145/1057977.1057980.

[11] H. T. Kung, J. T. Robinson, On optimistic methods for concurrency control, in: Proceedings of the Seventh International Conference on Very Large Data Bases (VLDB '81), 1981, pp. 351–364. doi:10.1145/582318.582322.

[12] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (1978) 558–565. doi:10.1145/359545.359563.

[13] D. Mishunov, True lies of optimistic user interfaces, Smashing Magazine, 2016. https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/.
[14] D. Meagher, Geometric modeling using octree encoding, Computer Graphics and Image Processing 19 (1982) 129–147. doi:10.1016/0146-664X(82)90104-6.
[15] Redis, Redis benchmarking, Redis documentation, 2026. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/.
[16] Redis, Redis pipelining, Redis documentation, 2026. https://redis.io/docs/latest/develop/use/pipelining/.