Geo-replication is an increasingly common requirement for computing services which desire low latency and high availability across the globe. In these services, replicas serve requests locally and independently. The lack of synchronisation across sites guarantees good performance but often comes at the cost of consistency; many recent research efforts, as a result, exhibit a race to re-define consistency and expressiveness in search of systems that are scalable and available. Most of these systems allow for independent, possibly conflicting, executions across geographically distinct sites, but all stop short of allowing locally conflicting operations: the local storage is viewed as a sequential black box. Using these systems then presents two main drawbacks. First, each replica, upon detecting write-write conflicts, greedily merges conflicting operations so as to never expose multiple values of an object and preserve the abstraction of a centralised, sequential system. Unfortunately, keeping the application isolated from the merging process means that the state post merge is more likely to break application invariants. But because of the merge, applications no longer have access to the full system history and cannot resolve the anomalies that arise. There is in practice no clean way for the storage to merge conflicting operations in a way that makes sense to the application. Second, there is a performance overhead associated with assuming the existence of strictly sequential local storage, when by contrast, databases rely on weaker isolation levels to improve performance.

To address these shortcomings, we present the design and implementation of TARDiS, a Git-like transactional key-value store which bridges the semantic gap introduced by building weakly consistent geo-replicated systems on top of local storage that is assumed to be sequential. Git recognises that multiple (possibly conflicting) branches can exist in parallel, and allows users to reconcile branches when/how they want to; TARDiS adopts a similar approach and explicitly tracks the distinct world views that emerge due to independent and concurrent operations. By adopting a Git-like view of the world and pushing weak consistency principles down to the local site, TARDiS improves local performance: conflicting operations no longer abort but fork the world on conflict. By allowing programmers to choose when/how to resolve conflicts, TARDiS removes the need for cumbersome programming restrictions, fixed resolution strategies, and expensive synchronisation. TARDiS is based on three design principles: (i) branching on conflict, (ii) maintaining intra-branch consistency and inter-branch isolation, and (iii) deferring the decisions of how and when to resolve conflicts (i.e. merge branches) to the application developer.

The first core principle of TARDiS is that transactional conflicts should be handled not through lock acquisition or transactional abort/rollback as is the norm in traditional transactional systems [8], but by branching the universe to create parallel world views, each reflecting a distinct execution path in the system. Branch-on-conflict naturally captures the reality of highly available distributed systems where replicas are allowed to process requests locally without coordination [14, 22, 24, 32, 38]. Across multiple sites, concurrent conflicting operations cannot be prevented and create implicit branches which break consistency. Branch-on-conflict allows TARDiS to expose those branches to the programmer. Branch-on-conflict further eliminates locks and abort/rollback from the performance critical path, satisfying the goals of highly available operations and addressing the NoSQL performance concerns without sacrificing transactional semantics.

The second core principle of the TARDiS design is to maintain intra-branch consistency and inter-branch isolation. Once a branch is formed due to a conflict, that branch should be kept isolated from other branches. Each branch then corresponds to a consistent execution path in the system. Without this aspect, reconciling conflicting operations is highly challenging, as there is no easy way for applications to preserve the intention of conflicting users. TARDiS is not the first system to allow for parallel multiversions of the same object [3, 15, 26, 38]. But, to the best of our knowledge, existing systems have systematically taken a strictly per object view of multiversions, ignoring transactions, cross-object consistency and therefore // inter-
branch isolation, making it extremely difficult to subsequently resolve conflicts.

The final core principle guiding the design of TARDiS is the following: whilst application semantics may expect a single world view, the storage system should not place any constraints on how or when divergent branches should be merged. TARDiS recognises that merging is fundamentally application specific and that its semantics cannot be handled generally by the storage. Moreover, by explicitly granting the choice of when to resolve conflicts, TARDiS allows conflict resolution to be delayed until convenient for the application. This delay can lead to more efficient systems (if merging is delayed until periods of low system load) and simpler resolution policies. Other systems have relied on the application developer/user to resolve conflicts [14, 26, 37], but they universally require the conflict to be resolved as soon as it is detected.

Our results show that a GIT-like transactional storage system is achievable with good performance and a simple interface. Our experience in developing applications on top of TARDiS shows that branch-on-conflict simplifies the subsequent merging: we demonstrate the benefits of branch isolation by porting a library of merging functions, specifically designed for weak consistency, to TARDiS: our implementation of CRDTs [32] results in 48% fewer lines of code than when built on top of non branch consistent storage.

In the rest of this paper, we first outline the design of TARDiS (§2), and introduce our new consistency model: branch consistency (§3). Next, we describe the system’s architecture (§5) and report on performance and application experiences (§6 and §7). Finally, we summarise related work (§8) and conclude (§9).

2 TARDiS Overview

With the three principles of branch on conflict, inter-branch isolation, and application-specific merge in mind, we design TARDiS, a transactional key-value store that can easily track the independent executions that arise in distributed systems. TARDiS consists of two conceptual pieces: (i) branch consistency, a new consistency model with support for branching semantics (we describe it more formally in Section 3), and (ii) explicit conflict tracking. Explicit conflict tracking is enabled by branch isolation and allows users to easily and efficiently navigate between branches. This in turn simplifies the design of arbitrary merging policies, and improves their performance (§7).

Our TARDiS prototype implements branch-on-conflict by extending multiversion concurrency control to provide both stale and parallel snapshots. Our design follows a two-layer architecture, as illustrated in Figure 1. The storage layer stores the independent versions of records and guarantees efficient modification/retrieval of objects; the consistency layer guarantees inter-branch isolation by explicitly keeping track of branches. Much of the TARDiS logic is geared towards efficiently mapping the consistency layer to the storage layer. Whilst TARDiS is designed to support replication, it is not intrinsically replicated. A separate replication service runs next to TARDiS and is responsible for disseminating committed transactions and applying remote transactions when appropriate.

3 System Model

We describe how branch consistency extends the traditional storage model to implement branch-on-conflict, guarantee inter-branch isolation and support arbitrary merging. For simplicity, we describe the model focusing on similarities and differences to Git.

Traditional Model  A storage system can be instantiated as a key-value store consisting of a set of objects Ω, each identified by a distinct key k. Each object is modeled as a deterministic state machine where a transition is induced by applying a write operation to the object. The current system state s is an assignment of values to all objects in the system.

A read of object o returns the current value v of that object. A write of v to object o induces a transition from a state value v_n to a state value v. A transaction t is a set of operations ordered by a partial order <_t, where all operations to the same object are ordered; the unique final operation in every completed transaction is either commit or abort [3]. Executing a transaction t induces a state transition from state s to state s’, where s’ is the result of applying the writes of t to s. The set T is the set of all transactions and forms a partial order <_T. Two transactions are concurrent if they are not ordered with respect to <_T. Transactions t_1 and t_2 conflict if changing their execution order (i) yields two distinct states s and s’ or (ii) changes one of their return value. Two transactions which do not conflict commute, we write: s + t_1 + t_2 = s + t_2 + t_1. The sequence of state transitions which the partial order of transactions <_T induces is referred to as an execution. The current state of the system is a state s such that ¬∃s’, s ≠ → s’. In otherwords, s is the most recent state of the system. An execution (<_e) is
valid iff (i) it is a linear extension of \(<_T\) (ii) it is totally ordered (aka sequential). An execution is sequential iff no concurrent transactions conflict; currently most distributed systems assume sequential storage and therefore disallow conflicting concurrent transactions.

**Git** Git [18] is a version control system which facilitates concurrent, possibly conflicting modifications to files. Users are allowed to “fork” (\(\text{git checkout -b }\) newbranch) the current repository state, and operate on their own branch before “merging” their view with others, possibly manually resolving conflicts. Developers can switch branches (\(\text{git checkout mybranch}\)) if desired, and navigate to past states if necessary.

**Branch Consistency** Branch consistency is a new consistency model for weakly consistent local storage which, like Git, allows for possibly conflicting operations to execute concurrently. Branch consistency explicitly tracks the independent executions that arise; as in Git, each transaction can choose which branch to execute from. But, contrary to Git: (i) branching is implicit rather than explicit: a Git user must explicitly request for a new branch to be created, whereas in branch consistency, a new branch is implicitly created when two conflicting transactions execute concurrently. Transactions do not have to be aware of other conflicting operations and never stall or abort (ii) branch consistency never forces merging. With Git, when a user tries to push modifications to a branch which has conflicting modifications, the user must first merge these modifications with its own operations, before committing or retrying. This approach is problematic in a geo-replicated system, as it requires expensive synchronisation across sites to determine whether concurrent transactions exist, and would cause transactions to stall or possibly abort: TARDiS ensures that users are never made aware of concurrent conflicting operations, unless they explicitly request to merge.

A transaction in branch consistency is associated with a constraint, which determines the state, or set of states from which it can execute. Regular transactions execute from a single state, whilst merge transactions execute from multiple concurrent states. A branch consistent execution \(e\) is a partially ordered set of transactions, such that, (i) it is an extension of \(<_T\), (ii) if two transactions \(t_1\) and \(t_2\) are concurrent and conflicting, then \(\exists n.s_1 \rightarrow \cdots \rightarrow s_2.\)

### 4 Interface

The TARDiS interface addresses two competing concerns: (i) minimising the increase in programming complexity (ii) providing sufficient flexibility to reason about concurrent branches of execution.

TARDiS is a transactional storage system which operates in two modes. The normal mode of operation is single
mode in which the programmer is allowed to (transactionally) read and write to a single branch. The experience of normal mode is exactly what programmers are accustomed to in traditional transactional systems, with the caveat that programmers must select a branch to operate on. The second mode of operation is merge mode in which the programmer selects two or more branches to read from and writes to a single state. The key difference between TARDiS merge mode and previous multi-value systems such as Dynamo [15] and Depot [26] is that TARDiS merge mode allows users to perform cross-object resolution atomically in order to maintain application invariants whilst previous systems were restricted to independently reconciling individual objects. The full TARDiS interface is shown in Table 1.

In branch isolation systems, begin and end constraints are fundamental for determining which branches a transaction will execute on. Intuitively, begin constraints select what states the transaction can read, whilst end constraints restrict the set of possible commit points. TARDiS supports the constraints in Table 2. An application which wishes to provide snapshot isolation and read-my-write guarantees would select the ancestor constraint as begin constraint and snapshot isolation as end constraint. To restrict branching, the application would instead specify the intersection of snapshot isolation and the no branching constraint. Using the Parent constraint as begin constraint gives clients the guarantee that they will operate on the same state to which they last committed.

5 TARDiS Design and Implementation

We next describe the design and implementation of TARDiS, focusing on the performance and space efficiency challenge of tracking multiple branches. We overview the base storage system and core data structures in Section 5.1. Section 5.2 describes single mode operation whilst Section 5.3 highlights the main differences of merge mode. Section 5.4 describes the garbage collection procedures and Section 5.5 sketches the default replication strategy.

5.1 Data and Metadata Storage

TARDiS is divided into two primary components: the consistency layer which tracks branches and states and the backend storage layer responsible for storing objects in the system.

5.1.1 Consistency Layer

The consistency layer in TARDiS tracks branches and leverages conflict dependencies to efficiently navigate the DAG. It stores a set of states. Each \( w \) corresponds to a distinct system snapshot and can be looked up by a unique identifier \( w.id \). The states are logically organised into a DAG corresponding to the phylogenetic history of the system.

The state DAG maintains the invariant that \( w.id > v.id \) for all children \( v \) of \( w \). Each state contains the write set of the transaction that produced the state, a link to its parent(s), and an ordered list of children. Each state additionally contains a fork path that concisely describes the historical evolution of that state.

A fork path consists of a series of fork points. Each fork point contains a state identifier \( i \) and branch number \( b \) indicating that the current state is a descendant of the \( b^{th} \) child of state \( i \). The combination of ordered state identifiers and explicit conflict tracking enables efficient descendant checking; dependency tracking and explicit path comparisons are very inefficient by comparison. Pseudocode for the descendant checking algorithm is shown in Figure 2. Figure 4 provides an example.

5.1.2 Storage layer

The storage layer stores objects written to TARDiS in records containing the value and identifier of the state produced when the record was written. Writes to the storage layer are persisted in a flat disk-backed b-tree indexed by a (key, worldId) tuple. A second, in-memory hashmap maintains a sorted list of version identifiers for each object.

Records for an individual object project onto the state DAG stored at the storage layer and are logically arranged into their own DAG structure. Example records are shown in Figure 4.

5.2 Basic Operation

In single-mode operation, transactions in TARDiS follow the following pattern. The user first begins a new transaction with a specified begin constraint; TARDiS identifies the most recent committed candidate parent state compatible with that constraint to use as the base parent. The user then issues a sequence of reads and writes to the system; reads return values visible at the parent state whilst writes are staged for addition into the storage layer b-tree. The overhead of operating across multiple branches
Single Mode | Merge Mode
---|---
√ | √
√ | √
√ | √
√ | √
√ | √
√ | √
√ | √
√ | √
√ | √

<table>
<thead>
<tr>
<th>Constraint</th>
<th>begin</th>
<th>end</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>√</td>
<td>√</td>
<td>Always Satisfies</td>
</tr>
<tr>
<td>Serializability</td>
<td>√</td>
<td></td>
<td>The transaction is serializable with this state</td>
</tr>
<tr>
<td>Snapshot Isolation</td>
<td>√</td>
<td></td>
<td>The transaction respects the snapshot isolation constraint with respect to this state</td>
</tr>
<tr>
<td>No Branching</td>
<td>√</td>
<td></td>
<td>This state has no children</td>
</tr>
<tr>
<td>K-Branching</td>
<td>√</td>
<td></td>
<td>This state has fewer than k-1 children</td>
</tr>
<tr>
<td>Parent</td>
<td>√</td>
<td>√</td>
<td>This state is the state to which the client last committed</td>
</tr>
<tr>
<td>Ancestor</td>
<td>√</td>
<td>√</td>
<td>This state is a descendant of the state to which the client last committed</td>
</tr>
<tr>
<td>World Identifier</td>
<td>√</td>
<td>√</td>
<td>This state ID matches the specified ID</td>
</tr>
<tr>
<td>Union</td>
<td>√</td>
<td>√</td>
<td>If at least one of the constrains is satisfied</td>
</tr>
<tr>
<td>Intersection</td>
<td>√</td>
<td>√</td>
<td>If all of the constrains are satisfied</td>
</tr>
</tbody>
</table>

Table 1: TARDiS API

is limited to branch selection at begin time.

5.2.1 Begin Transaction: State Selection

A transaction starts executing in TARDis by identifying a compatible state from which to read (the transaction’s candidate parent) (or multiple in the context of a merge transaction). In our implementation, a state is compatible if it satisfies the begin constraints and is the most recent state for this particular branch. The DAG in Figure 3 thus contains two compatible states, views 5 and 8 to which a newly executing transaction could append. In effect, this equivalent to allocating a consistent snapshot in snapshot isolation, where TARDis considers both state and parallel snapshots. TARDis appropriately identifies the correct view by conducting a breadth-first search through the DAG. The transaction is then associated with its candidate parent’s fork path.

```python
descendantCheck(x, y):
    if x.id == y.id then return True
    else if x.id > y.id then return False
    else if x.path ⊆ y.path then return False
    else return True
```

Figure 2: Pseudocode to check if state y is a descendant of x

5.2.2 Reading records

To read a record, a transaction accesses the in-memory key-version mapping structure which TARDis maintains, and processes each version ID in order, checking whether this identifier corresponds to a state that is on the same branch as the transaction’s current path. As the list of version IDs is topologically sorted, the first visible record observed by the transaction will always be the most recent one. We leverage the fact that state identifiers are monotonically increasing to cheaply maintain this topological ordering as a sorted list. Once a record identifier has been identified, the transaction directly accesses the on-disk storage and returns the record. Figure 4 provides a sample snapshot of the main system data structures: the state DAG, the fork paths associated with each state, and
the records. We consider three transactions A, B and C, associated with states 7,8 and 9 respectively. Transaction A would read v1 for key 1, v3 for key 2, v1 for key 3, and empty for key 4. Similarly, transaction B would read v1, v8, v1 and empty for keys 1 to 4. Finally, transaction C would read v1, v1, v9 and v9 for keys 1 to 4.

5.2.3 Writing values

Writing a value in TARDiS follows the same logic as get: the transaction first identifies the most recent visible record(s). It subsequently creates a new record version with these records listed as parents and the transaction’s state ID as version identifier. The record is then inserted into the disk-based b-tree and the in-memory-mapping updated to include the new version identifier. We note that this process is identical both in the presence or absence of write-write conflicts. Two conflicting transactions will identify the same parent record and create new records pointing to the same parent, forking the record structure.

5.2.4 Commit

Committing a transaction requires identifying the most recent parent state that is visible to the transaction, which may be more recent than its initial candidate parent. This is achieved through the *end constraint*. End constraints identify which concurrent committed transactions are not conflicting. In other words, starting from its candidate parent, the transaction “ripples” down the DAG to find the most recent committed parent to which the transaction can commit. The process is illustrated in Figure 5: views 9 and 11 commit off view 4 but do not conflict with the transaction that is about to commit. View 12, on the other hand does. The newly committed transaction thus commits after view 11.

**Read-only Transactions** Transactions which do not contain write-operations cannot induce conflicts. TARDiS does not persist them into the State DAG. This optimisation provides two benefits 1) it limits unnecessary DAG growth, which has memory overhead, removing pressure from the Garbage Collector 2) it removes the unnecessary overhead of identifying an appropriate commit location in the State DAG.

5.3 Merge Transactions

In TARDiS, merge operations are explicit and specify a set of branches to be reconciled through specifying a set of constraints. Merges are atomic and must reconcile all multivalues within the same transaction, collapsing them back to a single value. Implementation-wise, merge transactions are normal transactions with three exceptions 1) merge transactions select multiple parents, creating a state with multiple parents and record versions with multiple back pointers 2) merge transactions cannot walk down the DAG as in single mode 3) a number of additional operations are available in merge mode. *findMultivalues* (Table 1) navigates the DAG to list all conflicting keys, whilst *getCommonParent* uses the fork paths to quickly determine the last common state of two branches. Finally *getForID* uses the in-cache key-branch id mapping select the appropriate record for a particular state.
5.4 Garbage Collection

Traditional databases store only the active frontier of records, keeping space overheads manageable as 'old' transactions commit. This is in contrast to TARDiS, which, by default stores all stale/parallel versions and states. For performance and efficiency under finite storage, TARDiS implements an aggressive garbage collection policy, in three parts: (i) users place ceilings, which are promises not to use any previous states as a base state in the future (ii) a path compression algorithm compreses the State DAG to fork/merge points and current states (iii) a record promotion algorithm removes records which are no longer visible as a result of path-compression and ceilings.

Path Compression The State DAG compression relies on one core heuristic: most merging policies only require the fork points and the leaf states of a given execution. Above a ceiling, the state DAG can be compressed, removing all non-leaf and non-fork/merge nodes. This in effect reduces the DAG to explicitly tracking the nearest conflict dependency. This process is achieved in three stages (Figure 6): (i) a ceiling marking pass marks all nodes that are above a recently placed ceiling. Marked nodes cannot be selected as candidate parents. This ensures that no new transactions start above a ceiling. (ii) a safe-to-gc pass marks all nodes as safe to garbage collect if they are not currently selected as candidate parents, and for which all ancestors are also safe to garbage collect. This pass is necessary to avoid committing transactions rippling down deleted views, and ensures that a node will only be deleted after all its ancestors have been deleted (iii) a garbage collecting pass marks safe-to-gc nodes which are neither fork points nor merge points as garbage collectable. Garbage-collectable nodes are first “promoted” by mapping their state identifier to that of their most recent non-deleted descendant in a promotion table. All accesses looking up a deleted view will go to the promoted view instead. Once promoted, garbage collectable nodes can safely be removed from the state DAG. Node promotion is necessary to correlate record identifiers with deleted nodes. Consider object D in Figure 6, it is modified only once and will consequently have state identifier 1. All transactions that are descendants of state 1 will see object D. But, when state 1 is garbage collected, any transaction that tries to read object D and lookup the fork path of state 1 will fail. Promoting state 1 to state 8 ensures that that any such transaction will be redirected to state 8.

Record Promotion TARDiS’s iterative three-step garbage collection algorithm reduces the DAG significantly. Merge points specifically allows for fork points to be compressed. TARDiS must next delete record versions which will no longer be accessed, ensuring that the b-tree does not grow unbounded. As intermediate states are garbage collected, records which are associated with these deleted views are promoted to their first non-garbage collected descendants, whilst at the record storage level, records are updated to reflect their new identifier. This promotion creates large chains of records with the same state ID. The TARDiS get/put algorithm is such that only the most recent visible record will be returned: none of the promoted records which share an identifier will ever be accessed, bar the first one. They can therefore safely discarded. A full record promotion pass, as in path compression, reduces the record versions kept to current versions, conflicting records, and merges.

5.5 Replication

TARDiS is intended for use in loosely-coordinated distributed systems. In this context, transactions are replicated to all sites, but it is untenable to coordinate and ensure that all sites receive the same global order of transactions.

The replicator, in this context, simply disseminates the transactions to ensure that each transaction makes it to all other sites using a gossip protocol[1]. The transaction is accompanied with a parent constraint, indicating the parent world view to which the generated world view should be appended. Upon receiving a replicated transaction, the replicator applies it to its local storage (if possible). If not possible, it is cached to be appended later. Each transaction is guaranteed to be appended on the same world view at all sites.

Garbage collection is triggered by each local replicator, placing ceilings only after having received unanimous consent from all other replicators.

5.6 Implementation notes

Our prototype consists of 15K of Java code and uses the JDMB3 storage layer for record persistence to disk. We rely on Google Protobuffers v2.4.5 for message serialization and a custom implementation of the Netty networking library for inter-site communication. We find that accessing the individual records using the JDMB3 durability layer is currently a significant bottleneck for write-heavy workloads in our system and are investigating alternatives.
Figure 6: Path Compression Algorithm - A transaction places a ceiling above state 12. All ancestors of 9 are marked as safe-to-gc. State 10 has a non-empty candidate set, so cannot be marked as safe-to-gc. State 11 cannot be marked as safe-to-gc as its ancestors are not all safe-to-gc. All safe-to-gc nodes that are not merge points or fork points are marked as garbage collectable and deleted. This allows for more nodes to become garbage collectable (state 7), which are deleted on a subsequent pass.
Table 3: List of key value stores evaluated

<table>
<thead>
<tr>
<th>System</th>
<th>Transactions</th>
<th>Stale Versions</th>
<th>Branching</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeKV</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>SnapIsoKV</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>TARDiS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

6 Evaluation

Our goal in this evaluation is to quantify the performance of our TARDiS prototype and the performance tradeoffs associated with supporting branch consistency.

6.1 Evaluation Setup

Experiments are run on a shared local cluster of 2.67GHz Intel(R) Xeon(R) CPU X5650 machines connected over a 2Gbps network. Ping latencies between machines average 0.15 ms. To the best of our ability, all runs were performed in the absence of interfering workloads.

In addition to our TARDiS prototype, we evaluate three additional systems: (i) a non-transactional key-value store FreeKV (ii) a transactional key-value store SnapIsoKV supporting Snapshot Isolation, and (iii) BerkeleyDB v5.0.84 Java Edition. FreeKV and SnapIsoKV are simplified versions of the TARDiS codebase with restricted functionality as described in Table 3.

For benchmarking the system we configure a set of clients to issue non-interactive transactions consisting of four operations in a close loop. Read-only transactions contain 4 reads; read-write transactions contain 3 reads and a single write. We consider four transaction mixes differentiated by the ratio of read-only transactions to read-write transactions: Read-only (R-O, 100/0), Read-heavy (R-H, 75/25), and Write-heavy (W-H, 25/75).

Each experimental run lasts 120 seconds. We measure average throughput and latency starting 30 seconds into the run and stopping 15 seconds from the end.

Unless otherwise noted, runs with TARDiS have garbage collection enabled and use the any begin constraint and the snapshot isolation end constraint.

6.2 Microbenchmarks

In this section we perform a series of microbenchmarks designed to help understand the implications of various aspects of the TARDiS design.

6.2.1 Basic TARDiS performance

We begin by first measuring the performance of TARDiS using the three microbenchmark workloads described above; results are reported in Figure 7. The performance gap between the read-only workload and the other two workloads stems from the behavior of the JDMB3 library at the core of our record storage. Btree writes are fundamentally more expensive than reads and the periodic btree restructuring required due to the extensive multiversioning of TARDiS has a notable impact on performance in the presence of writes.

6.2.2 Benefits and Overheads Branching

In this section we attempt to quantify the overheads of transactions and branching by comparing TARDiS to FreeKV, a non-transactional key-value store, and SnapIsoKV, a traditional transactional key-value store supporting snapshot isolation. Figure 8 shows throughput-latency results for the three systems for the three workloads we consider.

In the read-only workload (Figure 8(a)), the performance of three systems are equivalent as the additional mechanisms in TARDiS and SnapIsoKV kick in only in the presence of contending writes.

In the read-heavy (Figure 8(b)) and write-heavy (Figure 8(c)) workloads, TARDiS exhibits a higher latency than SnapIsoKV and FreeKV due to the additional expenses of navigating the state DAG and the two stage record lookup. The throughput difference between TARDiS and SnapIsoKV is attributed to the fact that SnapIsoKV requires at least one of every pair of conflicting transactions to be aborted and re-executed while the branch isolation of TARDiS never requires transactions to be re-executed.
6.2.3 Garbage Collection

Without garbage collection, TARDiS accumulates every version of every object ever written, potentially leading to a very extensive state DAG and a large record storage. Figure 11 reports the throughput and latency for TARDiS with garbage collection both enabled and disabled. For the read-only workload, enabling garbage collection has a negative effect on performance due to the fact that read-only transactions do not create any extraneous states or records that can be garbage collected. For the other three workloads, garbage collection improves throughput by 15-33%. This improvement can be attributed to the removal of deprecated states from the state DAG and records from the btree, leading to more concise path definitions and more efficient accesses to the data structures.

Figure 11 shows the number of states generated with and without garbage collection and with and without read-only transactions. Unsurprisingly, enabling read-only transactions dramatically reduces the number of states created. The total number of states grows over time since this experiment does not merge divergent branches and fork points can not be garbage collected.

6.3 Replication

TARDiS scales linearly with the number of sites. Due to the cheapness of the ancestor constraint, replicated transactions do not take longer. There is no expensive computation incurred in doing the replication (computing up-stream effects). Our current implementation is a naive replicator with a single thread processing replicated transactions, as a result, replicated sites trail quite significantly.

6.4 Comparison with BerkeleyDB

To this point we have compared TARDiS with home-brewed systems to understand the tradeoffs of our design decisions. We now compare to BerkeleyDB, a simple commercially available database. The results shown in
Figure 11: Aggregated throughput as a function of number of replicas

Figure 12 indicate that our TARDiS prototype provides performance competitive to that of BerkeleyDB.

Table 4: Implemented CRDTs on SiKV and TARDiS, following algorithms in [31]

<table>
<thead>
<tr>
<th>Type</th>
<th>Op-Base</th>
<th>State-Base</th>
<th>L-O-C</th>
<th>TARDiS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-Counter</td>
<td>√</td>
<td></td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>Inc-Counter</td>
<td></td>
<td>√</td>
<td>162</td>
<td>123</td>
</tr>
<tr>
<td>PN-Counter</td>
<td></td>
<td>√</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>LWW Register</td>
<td></td>
<td>√</td>
<td>159</td>
<td>81</td>
</tr>
<tr>
<td>MVV Register</td>
<td></td>
<td>√</td>
<td>261</td>
<td>60</td>
</tr>
<tr>
<td>U-Set</td>
<td>√</td>
<td></td>
<td>190</td>
<td>111</td>
</tr>
<tr>
<td>U-Map</td>
<td></td>
<td>√</td>
<td>191</td>
<td>117</td>
</tr>
<tr>
<td>Or-Set</td>
<td>√</td>
<td></td>
<td>242</td>
<td>113</td>
</tr>
</tbody>
</table>

We implemented a subset of the CRDTs presented in [31], sufficient to design realistic applications [31], on top of both TARDiS and our snapshot isolation storage; Table 4 summarises each implementation. The smaller number of lines of code needed for TARDiS highlights the reduced implementation effort of building CRDTs on top of TARDiS. Two properties of TARDiS explain this simplification: (i) parent replication (ii) conflict tracking. Parent replication provides the guarantee that operations which locally execute on a given state will execute on the same state at all remote sites. Side-effects in operation-based CRDTs are needed to capture some amount of the local state. Parent replication removes this requirement and allows implementations to simply replicate the same operation. Similarly, the flat structure of traditional storage systems requires state-based CRDT implementations to keep track of state for all replicas at the time of executing the operation. This means that...
the size of the replicated objects, and sequentially replicated grows linearly with the number of replicas: the Counter CRDT is modeled as a vector containing an entry per replica, the multivalued register uses vector clocks to track causality. In effect, the developer must manually maintain a global view at each site, ensuring that it remains consistent as new operations arrive, whereas, by design, TARDiS efficiently tracks this global view.

8 Related Work

Formal Models In databases, serializability is the only isolation level which guarantees consistency and full ACID semantics [29]. Considered to be expensive, much work has gone into developing weaker isolation levels [47] and relaxing the notion of serializability using application-centric information [17]. Distributed systems have similarly obviated strong consistency (1-Copy Serializability [8], etc.) in favour of eventual consistency [41], but almost solely focus on the relationship between sites, assuming serializability of the local storage. These modes provide weak guarantees (ex: no transactions) often limited to guaranteeing eventual convergence. Other work provides slightly stronger guarantees at the cost of performance, with support for: transactions [24], consistent snapshots [34], and invariant preservation [22]. Both research avenues continue to assume strictly serializable storage.

Avoiding Conflicts The core challenge in geo-replicated systems lies in how to handle conflicts on different sites. One option is to preemptively ensures that conflicts do not happen. This can be achieved through strong synchronisation (two-phase locking [8] for example), optimised for specific workloads (short transactions in Sinfonia [4], long running transactions in Spanner [13]). Other techniques revolve around scheduling transactions in a conflict-avoidance fashion, using static analysis [43] or upon admission [40].

Redefining Conflicts For many applications, this cost is prohibitive. As such, many systems choose to redefine the notion of conflict, (i) either to something weaker (causal consistency [5,15], timeline consistency [12], parallel snapshot isolation [35], non-monotonic snapshot isolation [6], Pileus [39], etc.), (ii) to something application-specific: Bayou [18] allows users to attach code snippets to detect conflicts, or (iii) leverage specific operation properties to avoid synchronisation when possible (Red-Blue Consistency [22], Continuous Consistency Model [42], CRDTs [41], Escrow Transactions [33], MDCC [21]).

Resolving Conflicts When conflicts are allowed, systems must provide different resolution strategies to project conflicting execution onto sequential storage. COPS [25] provides a basic conflict resolution strategy of first-writer-wins, whilst Ficus [19], Dynamo [15], Depot [26], and Bayou all leave it up to the users (Bayou additionally assumes a total-order of writes, and undos/redo writes at each replica to match that order). Burckhardt et al. [9] associate specific resolution policies to objects, whilst operational transforms [36] leverage specific textual properties. All these resolution functions are per-object and do not support cross-object merging. By contrast, branch consistency allows programmers to see and resolve the entire state atomically, making it easier to enforce application invariants.

Branching Conflicts in a distributed system introduce implicit branching which must be reconciled when conflicts are detected. A number of systems expose this branching: version control systems (Git) allow users to operate on different branches; Olive [3] is a file-system which allows users to create/and fork snapshots efficiently, whilst ORI [28] tracks possibly divergent histories across multiple devices. They contrast with TARDiS as branching is explicit (git branch) rather than implicit as in our system. Some causally consistent systems allow for concurrent writes which can be exposed to the users (Ficus [19], Dynamo [15]). This is a limited notion of branching, which forks individual objects, not the whole world. These systems, as a result, do not provide any form of conflict tracking, and expose the programmer to multiple values. By contrast, in TARDiS, the programmer never has to deal with multivalued objects unless it explicitly requests it in merge mode. Concurrent Revisions [10] also fork state, but restrict themselves to a specifying DAG structure (semi-lattice) more reminiscent of the fork-join parallel system model than distributed systems. Unlike TARDiS, none provide transactions. The majority of work in branching takes place in the Byzantine context. SUNDR [23] develops fork consistency which isolates clients that see different values in a faulty server. Depot [27] uses the insight that faulty writes can be handled as concurrent writes, and extends the SUNDR model to support fork-joining (Fork Join Causal Consistency). Sporc [16] focuses on a client/server paradigm and uses operational transform to automatically join forks. FAUST uses similar techniques to define fork linearizability [11].
9 Conclusion

This paper introduced TARDiS, a Git-like transactional key value store targeted towards loosely-coupled distributed systems. By pushing weak consistency principles, namely, the admission of write-write conflicts, down to the local storage, TARDiS improves performance over sequential storage. By explicitly tracking concurrent branches and exposing distribution to the programmer, TARDiS simplifies application merging policies. Our results suggest that providing developers with a seemingly simple abstraction, that of a single, centralised convergent storage, is not appropriate for loosely coupled distributed systems. Applications need the gory details of distribution to make sense of the world in front of them.

References


