TARDiS: A branch and merge approach to weak consistency

By: Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvis, Allen Clement

Presented by: Samodya Abeysiriwardane
TARDiS

Transactional key-value store for weakly consistent systems
Weakly consistent systems

ALPS (Available, low Latency, Partition tolerance, high Scalability)

Conflicting operations cause replicas to diverge

Current solutions: Deterministic Writer Wins, per object eventual convergence (object as unit of merging)

Current solutions are not sufficient
Motivation

A wiki page with three objects

Edited at two georeplicated replicas
Motivation

Figure 1: Weakly-consistent Wikipedia
Main goal

Give applications access to context that is essential for reasoning about concurrent updates
Proposed solution

Expose branches as a unit of merging

- branch on conflict
- branch isolation
- application driven merges
Simple Example with Counters

Key value store of Counters
Need to define a merge function for the application

Merging two counters A and B

For counters 2-way merge

```javascript
fn merge (lca, a, b) = lca + (a-lca) + (b-lca)
```

For counters n-way merge

```javascript
fn merge {
    lca = find_fork_point
    val = lca
    for v in conflicting_values:
        val += (a - lca) + (b - lca)
}
```
```c
func increment(counter)
  Tx t = begin(AncestorConstraint)
  int value = t.get(counter)
  t.put(counter, value + 1)
  t.commit(SerializabilityConstraint)

func decrement(counter)
  Tx t = begin(AncestorConstraint)
  int value = t.get(counter)
  t.put(counter, value - 1)
  t.commit(SerializabilityConstraint)

func merge()
  Tx t = beginMerge(AnyConstraint)
  forkPoint = t.findForkPoints(t.parents).first
  int forkVal = t.getForID(counter, forkPt)
  list<int> currentVals =
    t.getForID(counter, t.parents)
  int result = forkVal
  foreach c in currentVals
    result += (c - forkVal)
  t.put(counter, result)
  t.commit(SerializabilityConstraint)
```

*Figure 3: TARDiS’ counter implementation*
Simple Example with Counter (Code)

**Client1**

T1:
inc(A, 3)

Tm:
merge

**Client2**

T2:
inc(B, 2)

Tm:
merge

T3:
inc(A, 5)
inc(B, 1)

Tm:
merge

13 = 5 (from S2) + (8-5)+(10-5)

10 = 9 (from S2) + (9-9)+(10-9)
Example

Impose an application invariant of
- if \( A > 8 \): \( B \) should max at 10
- the merge function can be changed to reflect that

Highlights the need for cross object merging semantics vs per object merging

Therefore branches as a unit of merging
Another example: Inventory

XYZ_stock: 1
ABC_stock: 3

Alice buys XYZ
XYZ_stock: 0

Bob buys XYZ and ABC
XYZ_stock: 0
ABC_stock: 2

Invariant: stock cannot be < 0

Merge
Bob gets XYZ, and exp
Alice gets error
XYZ_stock: 0
exp_stock: 2
Other advantages

No locking required

Branching as a fundamental abstraction for modeling conflicts end to end - replicas as well the local site can be viewed as branches
<table>
<thead>
<tr>
<th>S</th>
<th>M</th>
<th>return type</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>transaction</td>
<td>begin(beginConstraint)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>transaction</td>
<td>beginMerge(beginConstraint)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>void</td>
<td>put(key, value)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>value</td>
<td>get(key)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>value</td>
<td>getForID(key, StateID[])</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>key[]</td>
<td>findConflictWrites(StateID[])</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>forkPoints[]</td>
<td>findForkPoints(StateID[])</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>abort</td>
<td>commit</td>
</tr>
</tbody>
</table>

Table 2: TARDiS API - S: single mode, M: merge mode
Figure 2: TARDiS architecture
TARDiS architecture

Figure 2: TARDiS architecture
<table>
<thead>
<tr>
<th>Constraint</th>
<th>B</th>
<th>E</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>Guarantees Serializability</td>
</tr>
<tr>
<td>Snapshot Iso</td>
<td></td>
<td>√</td>
<td>Guarantees Snapshot Isolation</td>
</tr>
<tr>
<td>Read Committed</td>
<td></td>
<td>√</td>
<td>Guarantees Read Committed</td>
</tr>
<tr>
<td>No Branching</td>
<td>√</td>
<td></td>
<td>State has no children</td>
</tr>
<tr>
<td>K-Branching</td>
<td></td>
<td>√</td>
<td>State has fewer than k-1 children</td>
</tr>
<tr>
<td>Parent</td>
<td>√</td>
<td></td>
<td>State where client last committed</td>
</tr>
<tr>
<td>Ancestor</td>
<td>√</td>
<td></td>
<td>Child of client’s last committed state</td>
</tr>
<tr>
<td>State Identifier</td>
<td>√</td>
<td></td>
<td>State ID matches the specified ID</td>
</tr>
</tbody>
</table>

*Table 1: Begin (B) and end (E) constraints supported by TARDiS*
Consistency layer

begin(AncestorConstraint)
### Consistency layer

<table>
<thead>
<tr>
<th>Constraint</th>
<th>B</th>
<th>E</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>Guarantees Serializability</td>
</tr>
<tr>
<td>Snapshot Iso</td>
<td>✓</td>
<td>✓</td>
<td>Guarantees Snapshot Isolation</td>
</tr>
<tr>
<td>Read Committed</td>
<td>✓</td>
<td>✓</td>
<td>Guarantees Read Committed</td>
</tr>
<tr>
<td>No Branching</td>
<td>✓</td>
<td>✓</td>
<td>State has no children</td>
</tr>
<tr>
<td>K-Branching</td>
<td>✓</td>
<td>✓</td>
<td>State has fewer than k-1 children</td>
</tr>
<tr>
<td>Parent</td>
<td>✓</td>
<td>✓</td>
<td>State where client last committed</td>
</tr>
<tr>
<td>Ancestor</td>
<td>✓</td>
<td></td>
<td>Child of client’s last committed state</td>
</tr>
<tr>
<td>State Identifier</td>
<td>✓</td>
<td></td>
<td>State ID matches the specified ID</td>
</tr>
</tbody>
</table>

*Table 1: Begin (B) and end (E) constraints supported by TARDiS*
Consistency layer
Consistency layer

commit(SerializabilityConstraint)
Consistency layer

<table>
<thead>
<tr>
<th>Constraint</th>
<th>B</th>
<th>E</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>✓</td>
<td>✓</td>
<td>Always Satisfies</td>
</tr>
<tr>
<td><strong>Serializable</strong></td>
<td></td>
<td>✓</td>
<td>Guarantees Serializability</td>
</tr>
<tr>
<td>Snapshot Iso</td>
<td>✓</td>
<td></td>
<td>Guarantees Snapshot Isolation</td>
</tr>
<tr>
<td>Read Committed</td>
<td></td>
<td>✓</td>
<td>Guarantees Read Committed</td>
</tr>
<tr>
<td>No Branching</td>
<td>✓</td>
<td></td>
<td>State has no children</td>
</tr>
<tr>
<td>K-Branching</td>
<td></td>
<td>✓</td>
<td>State has fewer than k-1 children</td>
</tr>
<tr>
<td>Parent</td>
<td>✓</td>
<td></td>
<td>State where client last committed</td>
</tr>
<tr>
<td>Ancestor</td>
<td>✓</td>
<td></td>
<td>Child of client’s last committed state</td>
</tr>
<tr>
<td>State Identifier</td>
<td>✓</td>
<td></td>
<td>State ID matches the specified ID</td>
</tr>
</tbody>
</table>

*Table 1: Begin (B) and end (E) constraints supported by TARDiS*
TARDiS architecture

Figure 2: TARDiS architecture
Data structures

Key version mapping
A \mid S0

Record B-tree
A \mid S0

Fork paths:
The set of fork points
S0: {}
Data structures

Key version mapping
A | S0
B | S1
C | S1

Record B-tree
A | S0
B | S1
C | S1

Fork paths:
S0: {}
S1: {}
Data structures

Key version mapping
- A | S2, S0
- B | S1
- C | S1

Record B-tree
- A | S0 → S2
- B | S1
- C | S1

Fork paths: (set of tuples i,b where current state is bth child of state i)
- S0: {}
- S1: {}
- S2: { (1,1) }
Data structures

Key version mapping
A | S2, S0
B | S3, S1
C | S3, S1

Record B-tree
A | S0 → S2
B | S1 → S3
C | S1 → S3

Fork paths: (set of tuples i,b where current state is bth child of state i)
S0,S1: {}
S2: { (1,1) }
S3: { (1,2) }
A record version belongs to the selected branch if the fork path associated with this record version is a subset of the fork path of the transaction’s read state

```
1    descendantCheck(x, y):
2      if x.id = y.id then return true
3      else if x.id > y.id then return false
4      else if x.path ⊈ y.path then return false
5      else return true
```

*Figure 7: Check if state y can see records associated with state x*
If transaction read state is S3
Then which record version of C?
TARDiS architecture

Figure 2: TARDiS architecture
Evaluation setup

Shared local cluster
2.67 GHz Intel Xeon CPU X5650
48GB memory
2Gbps network

3 dedicated server machines
3 dedicated replicators
Equally spread clients
### Databases

**Berkley DB (BDB) - ACID datastore**

An implementation that does not require read write transactions to be verified against read-only transactions (OCC)

### Operation composition

**Read heavy (75R/25W)**

**Write heavy (0R/100W)**
Baseline TARDiS

Selecting constraints so that execution is serializable, and there is no branching

Figure 9: TARDiS-BDB vs BerkeleyDB vs OCC
With branching

(a) Uniform Read-Heavy

(b) Uniform Write-Heavy
With branching

Figure 10: Benefit of branching as a function of workload
CRDT implementations

Op-C: Operation Based Counter, PN-C: State Based Counter, LWW: Last-Writer-Wins Register, MV: Multivalued Register, Set: Or-Set
Branching as a means to provide an abstraction that lifts WW conflicts to the application level so that application developer can determine the intended outcome of conflicts in a weakly consistent application.
Hard for programmer to reason about the whole application state in merge function. Therefore have the ability to compose a merge function from multiple merge functions

Having the ability to push and pull from other states so that synchronization can happen asynchronously and by on request