Self-healing Data
Step by Step

Uwe Friedrichsen (codecentric AG) – NoSQL matters – Cologne, 29. April 2014
Why NoSQL?

- Scalability
- Easier schema evolution
- Availability on unreliable OTS hardware
- It’s more fun ...
Why NoSQL?

- Scalability
- Easier schema evolution
- Availability on unreliable OTS hardware
- It’s more fun ...
Challenges

- Giving up ACID transactions
- (Temporal) anomalies and inconsistencies
  short-term due to replication or long-term due to partitioning

It might happen. Thus, it will happen!
Consistency

ACID / 2PC

Strong Consistency
Quorum R&W / Paxos

Eventual Consistency
CRDT / Gossip / Hinted Handoff

Availability

Partition Tolerance
Strict Consistency (CA)

- Great programming model
  no anomalies or inconsistencies need to be considered
- Does not scale well
  best for single node databases

„We know ACID – It works!“

„We know 2PC – It sucks!“

Use for moderate data amounts
And what if I need more data?

- Distributed datastore
- Partition tolerance is a must
- Need to give up strict consistency (CP or AP)
Strong Consistency (CP)

- Majority based consistency model
can tolerate up to N nodes failing out of 2N+1 nodes
- Good programming model
  Single-copy consistency
- Trades consistency for availability
  in case of partitioning

*Paxos (for sequential consistency)*

*Quorum-based reads & writes*
Quorum-based reads & writes

- $N$ – number of nodes (replicas)
- $R$ – number of reads delivering the same result required
- $W$ – number of confirmed writes required

\[ W > N/2 \]
\[ R + W > N \]
Example 1

- 3 replica nodes
  
  This is: $N = 3$

- $W > N / 2 \Rightarrow W > 3 / 2 \Rightarrow W > 1.5$
  
  Let’s pick: $W = 2$  (can tolerate failure of 1 node)

- $R + W > N \Rightarrow R + 2 > 3 \Rightarrow R > 1$
  
  Let’s pick: $R = 2$  (can tolerate failure of 1 node)
R = 2 ✔

Client

2x ☐ / 1x ☐ ✔

Node 1
Node 2
Node 3
$R = 1 \; \times$  

Client

1x • / 1x ○  

Node 1

Node 2

Node 3
Example 2

• 3 replica nodes

  This is: $N = 3$

• $W > N / 2 \Rightarrow W > 3 / 2 \Rightarrow W > 1,5$

  Let’s pick: $W = 3$ \textit{(strict consistency – does not tolerate any failure)}

• $R + W > N \Rightarrow R + 3 > 3 \Rightarrow R > 0$

  Let’s pick: $R = 1$ \textit{(single read from any node sufficient)}
Example 3

• 5 replica nodes
  This is: \( N = 5 \)

• \( W > N / 2 \) \( \Rightarrow \) \( W > 5 / 2 \) \( \Rightarrow \) \( W > 2.5 \)
  Let’s pick: \( W = 3 \) (can tolerate failure of 2 nodes)

• \( R + W > N \) \( \Rightarrow \) \( R + 3 > 5 \) \( \Rightarrow \) \( R > 2 \)
  Let’s pick: \( R = 3 \) (can tolerate failure of 2 nodes)
Limitations of QB R&W

• Client-centric consistency
  Strong consistency only perceived on client, not on nodes

• Requires additional measures on nodes
  Nodes need to implement at least eventual consistency

• Requires fixed set of replica nodes
  Dynamic adding and removal of nodes not possible

*Use if client-perceived strong consistency is sufficient and simplicity trumps formal precision*
And what if I need more availability?

- Need to give up strong consistency (CP)
- Relax required consistency properties even more
- Leads to eventual consistency (AP)
Eventual Consistency (AP)

• Gives up some consistency guarantees
  no sequential consistency, anomalies become visible

• Maximum availability possible
  can tolerate up to N-1 nodes failing out of N nodes

• Challenging programming model
  anomalies usually need to be resolved explicitly

*Gossip / Hinted Handoffs*

*CRDT*
Conflict-free Replicated Data Types

- Eventually consistent, self-stabilizing data structures
- Designed for maximum availability
- Tolerates up to N-1 out of N nodes failing

*State-based CRDT: Convergent Replicated Data Type (CvRDT)*

*Operation-based CRDT: Commutative Replicated Data Type (CmRDT)*
A bit of theory first ...
Convergent Replicated Data Type

State-based CRDT – CvRDT

- All replicas (usually) connected
- Exchange state between replicas, calculate new state on target replica
- State transfer at least once over eventually-reliable channels
- Set of possible states form a Semilattice
  - Partially ordered set of elements where all subsets have a Least Upper Bound (LUB)
- All state changes advance upwards with respect to the partial order
Commutative Replicated Data Type

*Operation-based CRDT - CmRDT*

- All replicas (usually) connected
- Exchange update operations between replicas, apply on target replica
- Reliable broadcast with ordering guarantee for non-concurrent updates
- Concurrent updates must be commutative
That’s been enough theory ...
Counter
Op-based Counter

Data
    Integer i

Init
    i := 0

Query
    return i

Operations
    increment(): i := i + 1
    decrement(): i := i - 1
State-based G-Counter (grow only)
(Naïve approach)

\[\text{Data}\]
\[\text{Integer } i\]

\[\text{Init}\]
\[i := 0\]

\[\text{Query}\]
\[\text{return } i\]

\[\text{Update}\]
\[\text{increment(): } i := i + 1\]

\[\text{Merge}(j)\]
\[i := \text{max}(i, j)\]
State-based G-Counter (grow only)
(Naïve approach)
State-based G-Counter (grow only)
(Vector-based approach)

Data
Integer V[] / one element per replica set

Init
V := [0, 0, ... , 0]

Query
return ∑i V[i]

Update
increment(): V[i] := V[i] + 1 / i is replica set number

Merge(V')
∀i ∈ [0, n-1] : V[i] := max(V[i], V'[i])
State-based G-Counter (grow only)
(Vector-based approach)
State-based PN-Counter (pos./neg.)

- Simple vector approach as with G-Counter does not work
- Violates monotonicity requirement of semilattice
- Need to use two vectors
  - Vector P to track incements
  - Vector N to track decrements
  - Query result is $\sum_i P[i] - N[i]$
State-based PN-Counter (pos./neg.)

*Data*

Integer $P[\cdot], N[\cdot]$ / one element per replica set

*Init*

$P := [0, 0, \ldots, 0], N := [0, 0, \ldots, 0]$

*Query*

Return $\sum_i P[i] - N[i]$

*Update*

increment(): $P[i] := P[i] + 1$ / $i$ is replica set number
decrement(): $N[i] := N[i] + 1$ / $i$ is replica set number

*Merge($P', N'$)*

$\forall i \in [0, n-1]: P[i] := \max(P[i], P'[i])$
$\forall i \in [0, n-1]: N[i] := \max(N[i], N'[i])$
Non-negative Counter

Problem: How to check a global invariant with local information only?

- **Approach 1:** Only dec if local state is > 0
  - Concurrent decs could still lead to negative value

- **Approach 2:** Externalize negative values as 0
  - `inc(negative value) == noop()`, violates counter semantics

- **Approach 3:** Local invariant – only allow dec if \( P[i] - N[i] > 0 \)
  - Works, but may be too strong limitation

- **Approach 4:** Synchronize
  - Works, but violates assumptions and prerequisites of CRDTs
Sets
Op-based Set
(Naïve approach)

\textit{Data}

Set S

\textit{Init}

S := \{\}

\textit{Query(e)}

return \( e \in S \)

\textit{Operations}

\text{add}(e) : S := S \cup \{e\}

\text{remove}(e) : S := S \setminus \{e\}
Op-based Set
(Naïve approach)

\[ S = \{ e \} \]

\[ S = \{ e \} \]

\[ S = \{ e \} \]

\[ S = \{ e \} \]

\[ S = \{ e \} \]

\[ S = \{ e \} \]

\[ S = \{ e \} \]
State-based G-Set (grow only)

Data
Set S

Init
S := {} 

Query(e)
return e ∈ S 

Update
add(e): S := S U {e} 

Merge(S')
S = S U S'
State-based 2P-Set (two-phase)

**Data**
Set A, R  
/ A: added, R: removed

**Init**
A := {}, R := {}

**Query(e)**
return e ∈ A ∧ e ∉ R

**Update**
add(e): A := A ∪ {e}
remove(e): (pre query(e)) R := R ∪ {e}

**Merge(A', R')**
A := A ∪ A', R := R ∪ R'
Op-based OR-Set (observed-remove)

Data

Set $S$ / Set of pairs \{ (element $e$, unique tag $u$), ... \}

Init

$S := \{\}$

Query($e$)

return $\exists u : (e, u) \in S$

Operations

add($e$): $S := S \cup \{ (e, u) \}$ / $u$ is generated unique tag

remove($e$):

pre query($e$)

$R := \{ (e, u) \mid \exists u : (e, u) \in S \}$ /at source („prepare“)

$S := S \setminus R$ /downstream („execute“)
Op-based OR-Set (observed-remove)

R1

I
S = {}

add(e)
S = {e}

rmv(e)
S = {}

add(e_b)
S = {e_b}

R2

I
S = {}

add(e_b)
S = {e_b}

add(e_a)
S = {e_a, e_b}

rmv(e_a)
S = {e_b}

R3

I
S = {}

add(e)
S = {e_b}

More datatypes

- Register
- Dictionary (Map)
- Tree
- Graph
- Array
- List

plus more representations for each datatype
Garbage collection

- Sets could grow infinitely in worst case
- Garbage collection possible, but a bit tricky
  - Only remove updates that can be deleted safely and were received by all replicas
  - Usually implemented using vector clocks
  - Can tolerate up to n-1 crashes
  - Live only while no replicas are crashed
  - Can induce surprising behavior sometimes
  - Sometimes stronger consensus is needed (Paxos, ...)

...
Limitations of CRDTs

- Very weak consistency guarantees
  Strives for "quiescent consistency"

- Eventually consistent
  Not suitable for high-volume ID generator or alike

- Not easy to understand and model

- Not all data structures representable

*Use if availability is extremely important*
Further reading

1. Shapiro et al., Conflict-free Replicated Data Types, Inria Research report, 2011

2. Shapiro et al., A comprehensive study of Convergent and Commutative Replicated Data Types, Inria Research report, 2011


Wrap-up

- CAP requires rethinking consistency
- Strict Consistency
  ACID / 2PC
- Strong Consistency
  Quorum-based R&W, Paxos
- Eventual Consistency
  CRDT, Gossip, Hinted Handoffs

*Pick your consistency model based on your consistency and availability requirements*
The real world is not ACID

Thus, it is perfectly fine to go for a relaxed consistency model
@ufried