# High Availability under Eventual Consistency

Carlos Baquero
DEI, FEUP, Universidade do Porto

MEIC SDLE 2021

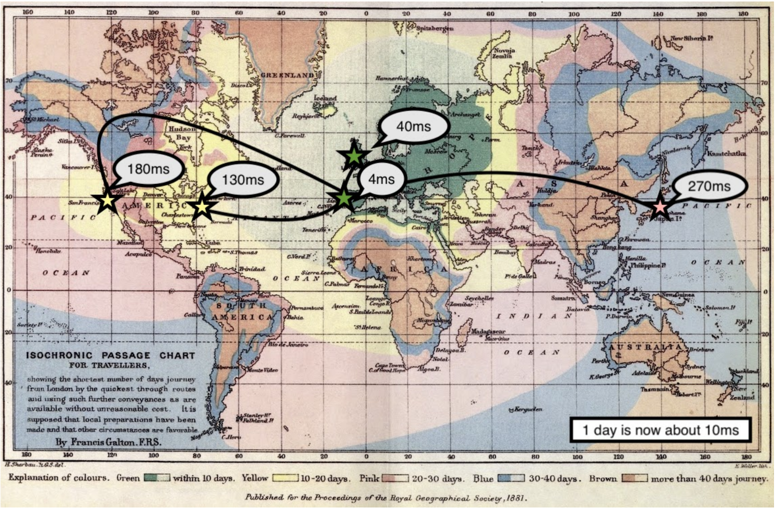
# The speed of communication in the 19th century
## Francis Galton Isochronic Map



# The speed of communication in the 21st century

RTT data gathered via `http://www.azurespeed.com`



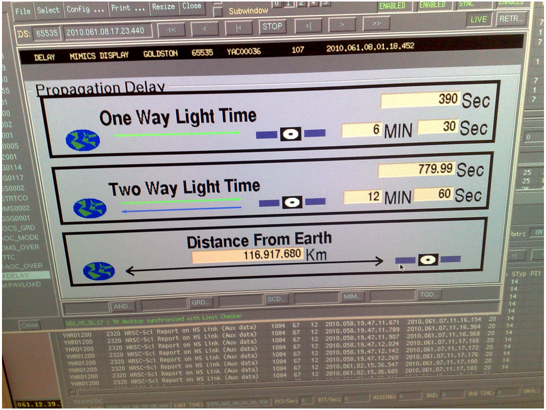


# The speed of communication in the 21st century
If you really like high latencies . . .



Time delay between Mars and Earth

`blogs.esa.int/mex/2012/08/05/time-delay-between-mars-and-earth/`




Delay/Disruption Tolerant Networking

`www.nasa.gov/content/dtn`

# Latency magnitudes
## Geo-replication



Carlos Baquero
DEI, FEUP, Universidade do Porto

- $\lambda$, up to 50ms (local region DC)
- $\Lambda$, between 100ms and 300ms (inter-continental)

### No inter-DC replication

Client writes observe $\lambda$ latency

### Planet-wide geo-replication

Replication techniques versus client side write latency ranges

Consensus/Paxos $[\Lambda, 2\Lambda]$ (with no divergence)

Primary-Backup $[\lambda, \Lambda]$ (asynchronous/lazy)

Multi-Master $\lambda$ (allowing divergence)

# EC and CAP for Geo-Replication

## Eventually Consistent. CACM 2009, Werner Vogels

- In an ideal world there would be only one consistency model: when an update is made all observers would see that update.
- Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

## CAP theorem. PODC 2000, Eric Brewer

Of three properties of shared-data systems – data consistency, system availability, and tolerance to network partition – only two can be achieved at any given time.

We will focus on AP.

# High Availability – Eventual Consistency



Carlos Baquero
DEI, FEUP, Universidade do Porto

A special case of weak consistency. After an update, if no new updates are made to the object, eventually all reads will return the same value, that reflects the last update. E.g: DNS.

This can later be reformulated to avoid quiescence, by adapting a session guarantee.

# Session Guarantees [Doug Terry, et al]



- Read Your Writes – read operations reflect previous writes.
- Monotonic Reads – successive reads reflect a non-decreasing set of writes.
- Writes Follow Reads – writes are propagated after reads on which they depend. (Writes made during the session are ordered after any Writes whose effects were seen by previous Reads in the session.)
- Monotonic Writes – writes are propagated after writes that logically precede them. (In other words, a Write is only incorporated into a server's database copy if the copy includes all previous session Writes.)

# From sequential to concurrent executions



Carlos Baquero
DEI, FEUP, Universidade do Porto

Consensus provides illusion of a single replica

This also preserves (slow) sequential behaviour

## Sequential execution

$$\mathit{Ops}\ O \qquad o \longrightarrow p \longrightarrow q$$

$$\mathit{Time} \qquad \dashrightarrow$$

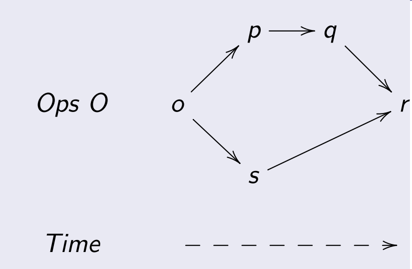
We have an ordered set $(O, <)$. $O = \{o, p, q\}$ and $o < p < q$

# From sequential to concurrent executions



EC Multi-master (or active-active) can expose concurrency

## Concurrent execution




Partially ordered set $(O, \prec)$. $o \prec p \prec q \prec r$ and $o \prec s \prec r$

Some ops in O are concurrent: $p \parallel s$ and $q \parallel s$

# Conflict-Free Replicated Data Types (CRDTs)



- Convergence after concurrent updates. Favor AP under CAP
- Examples include counters, sets, mv-registers, maps, graphs
- Operation based CRDTs. Operation effects must commute
- State based CRDTs are rooted on join semi-lattices

# Operation-based CRDTs, effect commutativity



Carlos Baquero
DEI, FEUP, Universidade do Porto

- In some datatypes all operations are commutative.
- PN-Counter: $\text{inc}(\text{dec}(c)) = \text{dec}(\text{inc}(c))$
- G-Set: $\text{add}_a(\text{add}_b(s)) = \text{add}_b(\text{add}_a(s))$

For more complex examples (e.g. sets with add and remove) operations need to generate "special" commutative effects. Here we will only cover examples of state-based CRDTs as they are more common in practice.

# State-based CRDTs, Join semi-lattices



- An (partial) ordered set S; $\langle S, \leq \rangle$.
- A join, $\sqcup$, deriving least upper bounds; $\langle S, \leq, \sqcup \rangle$.
- An initial state, usually the least element $\bot$; $\langle S, \leq, \sqcup, \bot \rangle$. $(\forall a \in S, a \sqcup \bot = a)$
- Alternative to a (unique) initial state, is a one time init in each replica assigning any element from $S$.
- Join properties in a semilattice $\langle S, \leq, \sqcup \rangle$:
  - Idempotence, $a \sqcup a = a$,
  - Commutativity, $a \sqcup b = b \sqcup a$,
  - Associative, $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$.
- $\leq$ reflects monotonic state evolution – increase of information.
- Updates must conform to $\leq$.
- In general, queries can return non-monotonic values, and in other domains than $S$. E.g: Returning a set size.

# Eventual Consistency, non stop



Carlos Baquero
DEI, FEUP, Universidade do Porto

Now convergence can related to known updates, with no need to stop updates: $\text{upds}(a) \subseteq \text{upds}(b) \Rightarrow a \leq b$.

This is slightly weaker than the previous definition and implies it: $\text{upds}(a) = \text{upds}(b) \Rightarrow a = b$.

# Design of Conflict-Free Replicated Data Types



Carlos Baquero
DEI, FEUP,
Universidade do Porto

A partially ordered log (polog) of operations implements any CRDT

Replicas keep increasing local views of an evolving distributed polog

Any query, at replica $i$, can be expressed from local polog $O_i$

Example: Counter at $i$ is $|\{\mathsf{inc} \mid \mathsf{inc} \in O_i\}| - |\{\mathsf{dec} \mid \mathsf{dec} \in O_i\}|$

CRDTs are efficient representations that follow some general rules

# Principle of permutation equivalence

If operations in sequence can commute, preserving a given result, then under concurrency they should preserve the same result

## Sequential

$$\mathrm{inc}(10) \longrightarrow \mathrm{inc}(35) \longrightarrow \mathrm{dec}(5) \longrightarrow \mathrm{inc}(2)$$

$$\mathrm{dec}(5) \longrightarrow \mathrm{inc}(2) \longrightarrow \mathrm{inc}(10) \longrightarrow \mathrm{inc}(35)$$

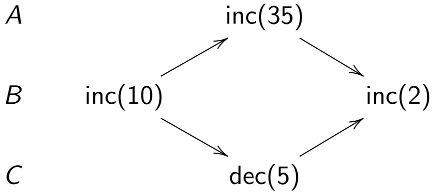
## Concurrent

$$\begin{tikzcd}
 & \mathrm{inc}(35) \arrow[dr] & \\
\mathrm{inc}(10) \arrow[ur] \arrow[dr] & & \mathrm{inc}(2) \\
 & \mathrm{dec}(5) \arrow[ur] &
\end{tikzcd}$$

You guessed: Result is 42

# Implementing Counters

## Example: CRDT PNCounters




Lets track total number of incs and decs done at each replica

$$\{A(\text{incs}, \text{decs}), \ldots, C(\ldots, \ldots)\}$$

# Implementing Counters
## Example: CRDT PNCounters



Separate positive and negative counts are kept per replica

$$
\begin{tikzcd}
A & & \{A(35,0), B(10,0)\} \arrow[dr] & \\
B & \{B(10,0)\} \arrow[ur] \arrow[dr] & & \{A(35,0), B(12,0), C(0,5)\} \\
C & & \{B(10,0), C(0,5)\} \arrow[ur] &
\end{tikzcd}
$$

Joining does point-wise maximums among entries (semilattice)

At any time, counter value is sum of incs minus sum of decs

# State-based CRDTs: PN-Counter



Carlos Baquero
DEI, FEUP, Universidade do Porto

$$
\begin{aligned}
\Sigma &= I \rightarrow \mathbb{N} \times \mathbb{N} \\
\sigma_i^0 &= \{(r, (0,0)) \mid r \in I\} \\
\mathrm{apply}_i(\mathrm{inc}, m) &= m\{i \mapsto (\mathrm{fst}(m(i)) + 1, \mathrm{snd}(m(i)))\} \\
\mathrm{apply}_i(\mathrm{dec}, m) &= m\{i \mapsto (\mathrm{fst}(m(i)), \mathrm{snd}(m(i)) + 1)\} \\
\mathrm{eval}_i(\mathrm{rd}, m) &= \sum_{r \in I} \mathrm{fst}(m(r)) - \mathrm{snd}(m(r)) \\
\mathrm{merge}_i(m, m') &= \{(r, \max(m(r), m'(r))) \mid r \in I\}
\end{aligned}
$$

Note: max is pointwise maximum

# Registers

Registers are an ordered set of write operations

## Sequential execution

$$\begin{tikzcd} A & \mathrm{wr}(x) \arrow[r] & \mathrm{wr}(j) \arrow[r] & \mathrm{wr}(k) \arrow[r] & \mathrm{wr}(x) \end{tikzcd}$$

## Sequential execution under distribution

$$\begin{tikzcd} A & \mathrm{wr}(x) \arrow[dr] & & & \mathrm{wr}(x) \\ B & & \mathrm{wr}(j) \arrow[r] & \mathrm{wr}(k) \arrow[ur] & \end{tikzcd}$$

Register value is $x$, the last written value

# Last Writer Wins



Carlos Baquero
DEI, FEUP, Universidade do Porto

A simple approach to evolve state without strong coordination, is to adopt a *Last Writer Wins* policy (see also Thomas write rule). Recently popularized in the Cassandra system, this policy uses timestamps to discard *older* writes and attain convergence.

$$
\begin{aligned}
\Sigma &= T \times \mathbb{N} \\
\sigma_i^0 &= (0,0) \\
\mathrm{apply}_i((\mathrm{wr}, t', n'), (t, n)) &= (t, n) \textbf{ if } t' < t \textbf{ else } (t', n') \\
\mathrm{eval}_i(\mathrm{rd}, (t, n)) &= n \\
\mathrm{merge}_i((t, n), (t', n')) &= (t, n) \textbf{ if } t' < t \textbf{ else } (t', n')
\end{aligned}
$$

LWW Integer

# Implementing Registers
## *Naive* Last-Writer-Wins



CRDT register implemented by attaching local wall-clock times

**Sequential execution under distribution**

$$\begin{tikzcd}[column sep=small]
A & (11{:}00)x \arrow[dr] & & & (11{:}30)? \arrow[dr] & \\
B & & (12{:}02)j \arrow[r] & (12{:}05)k \arrow[ur] & & ?
\end{tikzcd}$$

Problem: Wall-clock on B is one hour ahead of A

Value $x$ might not be writeable again at A since 12:05 $>$ 11:30

# Registers
## Sequential Semantics



Register shows value $v$ at replica $i$ iff

$$\mathsf{wr}(v) \in O_i$$

and

$$\nexists \mathsf{wr}(v') \in O_i \cdot \mathsf{wr}(v) < \mathsf{wr}(v')$$

# Preservation of sequential semantics



Carlos Baquero
DEI, FEUP, Universidade do Porto

Concurrent semantics should preserve the sequential semantics

This also ensures correct sequential execution under distribution

# Multi-value Registers

Concurrency semantics shows all concurrent values

$$\{v \mid \mathsf{wr}(v) \in O_i \wedge \nexists \mathsf{wr}(v') \in O_i \cdot \mathsf{wr}(v) \prec \mathsf{wr}(v')\}$$

## Concurrent execution

$$\begin{tikzcd}
A & \mathsf{wr}(x) \arrow[r] \arrow[dr] & \mathsf{wr}(y) \arrow[rr] & & \{y,k\} \arrow[r] & \mathsf{wr}(m) \arrow[r] & \{m\} \\
B & & \mathsf{wr}(j) \arrow[r] & \mathsf{wr}(k) \arrow[ur] & & &
\end{tikzcd}$$

Dynamo shopping carts are multi-value registers with payload sets

The $m$ value could be an application level merge of values $y$ and $k$

# Implementing Multi-value Registers

Concurrency can be precisely tracked with version vectors

## Concurrent execution (version vectors)

$$
\begin{tikzcd}
A & {[1,0]x} \arrow[r] \arrow[dr] & {[2,0]y} \arrow[rr] & & {[2,0]y, [1,2]k} \arrow[r] & {[3,2]m} \\
B & & {[1,1]j} \arrow[r] & {[1,2]k} \arrow[ur] & &
\end{tikzcd}
$$

Metadata can be compressed with a common causal context and a single scalar per value (dotted version vectors)

# Registers in Redis
## LWW arbitration



Multi-value registers allows executions leading to concurrent values

Presenting concurrent values is at odds with the sequential API

Redis both tracks causality and registers wall-clock times

Querying uses Last-Writer-Wins selection among concurrent values

This preserves correctness of sequential semantics

A value with clock 12:05 can still be causally overwritten at 11:30

# State-based CRDTs: G-Set

$$
\begin{aligned}
\Sigma &= \mathcal{P}(V) \\
\sigma_i^0 &= \{\} \\
\mathsf{apply}_i((\mathsf{add}, v), s) &= s \cup \{v\} \\
\mathsf{eval}_i(\mathsf{rd}, s) &= s \\
\mathsf{merge}_i(s, s') &= s \cup s'
\end{aligned}
$$

# State-based CRDTs: 2P-Set



$$
\begin{aligned}
\Sigma &= \mathcal{P}(V) \times \mathcal{P}(V) \\
\sigma_i^0 &= \{\}, \{\} \\
\mathsf{apply}_i((\mathsf{add}, v), (s, t)) &= s \cup \{v\}, t \\
\mathsf{apply}_i((\mathsf{rmv}, v), (s, t)) &= s, t \cup \{v\} \\
\mathsf{eval}_i(\mathsf{rd}, s) &= s \setminus t \\
\mathsf{merge}_i((s, t), (s', t')) &= s \cup s', t \cup t'
\end{aligned}
$$

Consider add and rmv operations

$$X = \{\ldots\},\ \mathsf{add(a)} \longrightarrow \mathsf{add(c)} \text{ we observe that } \mathsf{a, c} \in \mathsf{X}$$

$$X = \{\ldots\},\ \mathsf{add(c)} \longrightarrow \mathsf{rmv(c)} \text{ we observe that } \mathsf{c} \notin \mathsf{X}$$

In general, given $O_i$, the set has elements

$$\{e \mid \mathsf{add(e)} \in \mathsf{O_i} \land \nexists \mathsf{rmv(e)} \in \mathsf{O_i} \cdot \mathsf{add(e)} < \mathsf{rmv(e)}\}$$

2P-Set breaks sequential semantics

# Sets
## Concurrency Semantics

Problem: Concurrently adding and removing the same element

### Concurrent execution

$$
\begin{tikzcd}
A & \mathrm{add}(x) \arrow[r] \arrow[dr] & \mathrm{rmv}(x) \arrow[rr] & & \{?\} \arrow[r] & \mathrm{add}(x) \arrow[r] & \{x\} \\
B & & \mathrm{rmv}(x) \arrow[r] & \mathrm{add}(x) \arrow[ur] & & &
\end{tikzcd}
$$

Carlos Baquero
DEI, FEUP, Universidade do Porto

Let's choose Add-Wins

Consider a set of known operations $O_i$, at node $i$, that is ordered by an *happens-before* partial order $\prec$. Set has elements

$$\{e \mid \mathsf{add(e)} \in \mathsf{O_i} \ \wedge \nexists \ \mathsf{rmv(e)} \in \mathsf{O_i} \cdot \mathsf{add(e)} \prec \mathsf{rmv(e)}\}$$

Carlos Baquero
DEI, FEUP, Universidade do Porto

Let's choose Add-Wins

Consider a set of known operations $O_i$, at node $i$, that is ordered by an *happens-before* partial order $\prec$. Set has elements

$$\{e \mid \mathsf{add(e)} \in \mathsf{O_i} \ \wedge \nexists\, \mathsf{rmv(e)} \in \mathsf{O_i} \cdot \mathsf{add(e)} \prec \mathsf{rmv(e)}\}$$

Is this familiar?

# Concurrency Semantics
## Add-Wins Sets



Carlos Baquero
DEI, FEUP, Universidade do Porto

Let's choose Add-Wins

Consider a set of known operations $O_i$, at node $i$, that is ordered by an *happens-before* partial order $\prec$. Set has elements

$$\{e \mid \mathsf{add(e)} \in \mathsf{O_i} \ \wedge \nexists\, \mathsf{rmv(e)} \in \mathsf{O_i} \cdot \mathsf{add(e)} \prec \mathsf{rmv(e)}\}$$

Is this familiar?

The sequential semantics applies identical rules on a total order

# Concurrency Semantics
## Add-Wins Sets



Carlos Baquero
DEI, FEUP, Universidade do Porto

Let's choose Add-Wins

Consider a set of known operations $O_i$, at node $i$, that is ordered by an *happens-before* partial order $\prec$. Set has elements

$$\{e \mid \mathsf{add(e)} \in \mathsf{O_i} \ \wedge \nexists\ \mathsf{rmv(e)} \in \mathsf{O_i} \cdot \mathsf{add(e)} \prec \mathsf{rmv(e)}\}$$

Is this familiar?

The sequential semantics applies identical rules on a total order

Redis CRDT sets are Add-Wins Sets

# State-based CRDTs: Basic Add-Wins Set

## Observed-remove add-wins



Carlos Baquero
DEI, FEUP, Universidade do Porto

$$
\begin{aligned}
\Sigma &= \mathcal{P}(T \times V) \times \mathcal{P}(T) \\
\sigma_i^0 &= \{\}, \{\} \\
\mathrm{apply}_i((\mathrm{add}, v), (s, t)) &= s \cup \{(utag(), v)\}, t \\
\mathrm{apply}_i((\mathrm{rmv}, v), (s, t)) &= s, t \cup \{u \mid (u, v) \in s\} \\
\mathrm{eval}_i(\mathrm{rd}, s) &= \{v \mid (u, v) \in s \wedge u \notin t\} \\
\mathrm{merge}_i((s, t), (s', t')) &= s \cup s', t \cup t'
\end{aligned}
$$

# Equivalence to a sequential execution?

## Add-Wins Sets

Can we always explain a concurrent execution by a sequential one?

### Concurrent execution

$$\begin{array}{ll} A & \{x,y\} \longrightarrow \mathrm{add}(y) \longrightarrow \mathrm{rmv}(x) \longrightarrow \{y\} \longrightarrow \{x,y\} \\ B & \{x,y\} \longrightarrow \mathrm{add}(x) \longrightarrow \mathrm{rmv}(y) \longrightarrow \{x\} \longrightarrow \{x,y\} \end{array}$$

### Two (failed) sequential explanations

$$\begin{array}{ll} H1 & \{x,y\} \longrightarrow \ldots \longrightarrow \mathrm{rmv}(x) \longrightarrow \{\cancel{x},y\} \\ H2 & \{x,y\} \longrightarrow \ldots \longrightarrow \mathrm{rmv}(y) \longrightarrow \{x,\cancel{y}\} \end{array}$$

Concurrent executions can have richer outcomes

Alternative: Let's choose Remove-Wins

$$X_i \doteq \{e \mid \mathsf{add(e)} \in \mathsf{O_i} \;\wedge \forall\, \mathsf{rmv(e)} \in \mathsf{O_i} \cdot \mathsf{rmv(e)} \prec \mathsf{add(e)}\}$$

Alternative: Let's choose Remove-Wins

$$X_i \doteq \{e \mid \mathrm{add(e)} \in \mathrm{O_i} \ \wedge \forall \ \mathrm{rmv(e)} \in \mathrm{O_i} \cdot \mathrm{rmv(e)} \prec \mathrm{add(e)}\}$$

Remove-Wins requires more metadata than Add-Wins

Both Add and Remove-Wins have same semantics in a total order

They are different but both preserve sequential semantics

# Overview. Delaying choice of semantics



Thought experiment: A CRDT Set data type could store enough information to allow a parametrized query that shows both Add-Wins or Remove-Wins. One can expect a metadata size penalty for the flexibility.

# Sequence/List

## Weak/Strong Specification [Attiya et al, PODC 16]



### Element $x$ is kept

$$\begin{tikzcd}
 & & & \text{rpush(b)} \arrow[r] & \langle xb \rangle \arrow[dr] & \\
\langle\rangle \arrow[r] & \text{lpush(x)} \arrow[r] & \langle x \rangle \arrow[ur] \arrow[dr] & & & \langle axb \rangle \\
 & & & \text{lpush(a)} \arrow[r] & \langle ax \rangle \arrow[ur] &
\end{tikzcd}$$

### Element $x$ is removed (Redis enforces Strong Specification)

$$\begin{tikzcd}
 & & & \text{rpush(b)} \arrow[r] & \langle xb \rangle \arrow[dr] & & \\
\langle\rangle \arrow[r] & \text{lpush(x)} \arrow[r] & \langle x \rangle \arrow[ur] \arrow[r] \arrow[dr] & \text{rem(x)} \arrow[r] & \langle\rangle \arrow[r] & \langle ab \rangle & \neg\langle ba \rangle \\
 & & & \text{lpush(a)} \arrow[r] & \langle ax \rangle \arrow[ur] & &
\end{tikzcd}$$

# Take home message



- Concurrent executions are needed to deal with latency
- Behaviour changes when moving from sequential to concurrent

Road to accommodate transition:

- Permutation equivalence
- Preserving sequential semantics
- Concurrent executions lead to richer outcomes

CRDTs provide sound guidelines and encode policies