Distributed object storage is centralised

A quest for autonomy in the modern hosting ecology

Adrien Luxey

Wednesday, 28th April, 2021

A very casual motivation

I want to host **resilient web services** with **acceptable performance** on commodity hardware behind **household networks**.

Keywords

- Decentralised networks
- Edge computing
- Distributed storage
- Privacy

Context

Resilience: Ability to recover quickly from failures and changes.

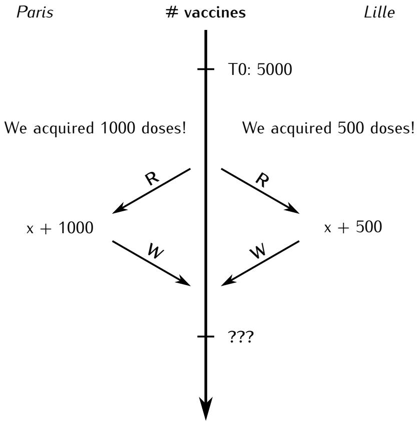
Only achievable through distribution of the hosted applications across several physical locations.

Application = computations on data

- **Computation**: Stateless; easy to distribute & orchestrate.
- **Data**: Stateful; hard to distribute & full of trade-offs.

Concurrent writes example

How to lose vaccines



The problem

Can we design an available data store tailored for adverse network conditions?

The CAP theorem

Eric Brewer's theorem

"A shared-state system can have **at most two** of the following properties at any given time:

- **C**onsistency
- **A**vailability
- **P**artition tolerance"

Under network partitions, a distributed data store has to sacrifice either availability or consistency.

- **Consistency-first**: Abort incoming queries;
- **Availability-first**: Return possibly stale data.

Consistency-first: the ACID model

Transaction: unit of work within an ACID data store.

- **Atomicity**: Transactions either complete entirely or fail.
  No transaction ever seen as in-progress.
- **Consistency**: Transactions always generate a valid state.
  The database maintains its invariants across transactions.
- **Isolation**: Concurrent transactions are seen as sequential.
  Transactions are serializable, or sequentially consistent.
- **Durability**: Committed transactions are never forgotten.

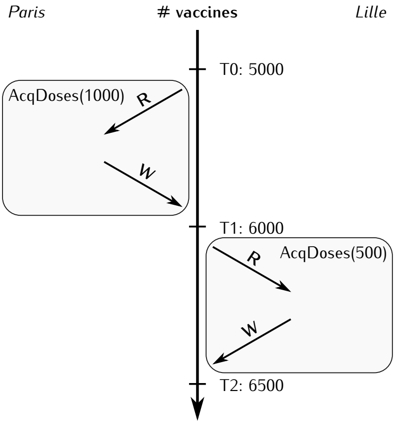
Reads are fast, writes are slow.

Example: relational databases.

Concurrent writes in ACID


```
transaction AcqDoses(y):
  x <- SELECT #vaccines;
  UPDATE #vaccines = (x + y);
```

Supports compound operations.



Availability-first: the BASE model

Some apps prefer availability, e.g. Amazon products' reviews.

The BASE model trades Consistency & Isolation for Availability.

- **Basic Availability**: The data store thrives to be available.
- **Soft-state**: Replicas can disagree on the valid state.
- **Eventual consistency**: In the absence of write queries, the data store will eventually converge to a single valid state.

Writes are fast, reads are slow.

Examples: key-value & object stores.

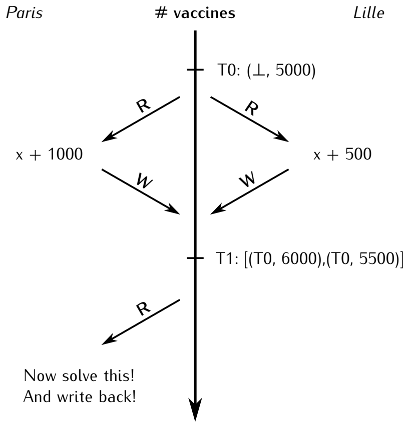
Concurrent writes in BASE

Object

- Unique key
- Arbitrary value
- Metadata

Conflict resolution = client's job!

No compound operations.

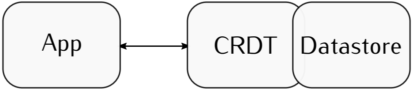


Strong Eventual Consistency w/ CRDTs

M. Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg, 2011

Strong Eventual Consistency (SEC)

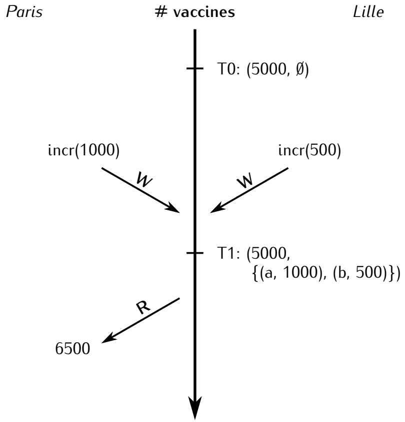
- CRDTs specify distributed operations
- Conflicts will be solved according to specification
- Proven & bound eventual convergence



Concurrent writes with CRDTs


```
CRDT Counter(x0):
  history = {}
  op. incr(y):
    history ∪= {(UUID(), y)}
  op. decr(y):
    history ∪= {(UUID(), -y)}
  op. read():
    x = x0
    for (_, y) in history:
        x += y
    return x
```

Operations commute?
$\implies$ screw total order!



A complex CRDT: the DAG

payload set V, A -- *sets of pairs* { (element e, unique-tag w), ... }
  initial $\varnothing, \varnothing$ -- *V: vertices; A: arcs*
query *lookup* (vertex v) : boolean b
  let $b = (\exists w : (v, w) \in V)$
query *lookup* (arc (v', v'')) : boolean b
  let $b = (lookup(v') \wedge lookup(v'') \wedge (\exists w : ((v', v''), w) \in A)$
update *addVertex* (vertex v)
  prepare (v) : w
    let $w = unique()$ -- *$unique()$ returns a unique value*
  effect (v, w)
    $V := V \cup \{(v, w)\}$ -- *v + unique tag*
update *removeVertex* (vertex v)
  prepare (v) : R
    pre $lookup(v)$ -- *precondition*
    pre $\nexists v' : lookup((v, v'))$ -- *v is not the head of an existing arc*
    let $R = \{(v, w) | \exists w : (v, w) \in V\}$ -- *Collect all unique pairs in V containing v*
  effect (R)
    $V := V \setminus R$
update *addArc* (vertex v', vertex v'')
  prepare (v', v'') : w
    pre $lookup(v')$ -- *head node must exist*
    let $w = unique()$ -- *$unique()$ returns a unique value*
  effect (v', v'', w)
    $A := A \cup \{((v', v''), w)\}$ -- *(v', v'') + unique tag*
update *removeArc* (vertex v', vertex v'')
  prepare (v', v'') : R
    pre $lookup((v', v''))$ -- *$arc(v', v'')$ exists*
    let $R = \{((v', v''), w) | \exists w : ((v', v''), w) \in A\}$
  effect (R) -- *Collect all unique pairs in A containing arc (v', v'')*
    $A := A \setminus R$

A complex CRDT: the DAG

Just to say I swept a lot under the rug.

For details, go read:
M. Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg, 2011

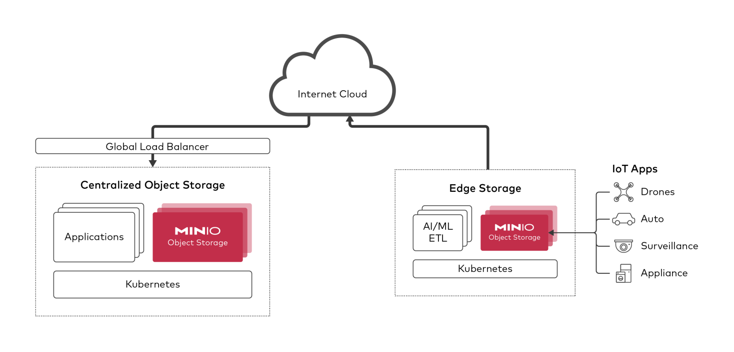
For an implementation, check **AntidoteDB**.

State of the practice

Path dependency to the "cloud"

The BASE model is fashionable because

"*High-performance* object storage for *AI analytics* with PBs of *IoT data streams* at the *edge*, using *5G*."



- Always backed by cloud: high performance network links.
- Edge nodes always seen as clients or data sources, not peers.

Why?

- **Privacy**: no prying eyes besides your ISP
- **Control** of your infrastructure
- **Ecology**: reuse old hardware

Tim Berners-Lee (1994)

"Now, if someone tries to monopolize the Web, for example pushes proprietary variations on network protocols, then that would make me unhappy."

- Make Tim Berners-Lee happy

What?

A data store for commodity hardware on heterogenous household connections.

Targetting user-facing services

- Static sites
- E-mails
- Instant communication
- Collaboration

Nothing fancy like sensors data streams, AI or IoT.

What?

Requirements

- **No single point of failure** / flat hierarchy:
  Any node can die for extended periods of time.
- **Multi-site**: cluster spans regions/countries.
- **Acceptable performance**.
- **Lightweight**: targets legacy hardware.
- **Conceptually simple**: built for low-tech organisations.
  Adding/maintaining cluster nodes should be easy.

Non-goals

- **Super badass performance**.
- NAT **traversal** etc.: we require full-mesh connectivity.

How?

- Theoretically possible with object storage & CRDTs.
- Household uplinks are getting decent (optical fibers).

Research Questions

- Decent performance despite bad inter-node connectivity.
- Tailoring workloads as a function of nodes' capabilities:
  - Make use of low-end nodes (e.g. Raspberry Pis),
  - Avoid impeding global performance because of low-end nodes.
- Building CRDTs for target use-cases:
  - Software engineering: DSL or native code?
  - Provide APIs to data store users? Risky?
- Cluster management: effortless UX, low perf. overhead.

Brought to you by the Deuxfleurs association

deuxfleurs.fr – a libre hosting association with a vision

"Shifting the current structure of the Internet from a world of a few very large service providers, to a world where services are hosted by a variety of smaller organisations."

Our goals

- To propose performant & reliable libre services for the masses
- To host and administer our infrastructure ourselves
- To allow members to contribute storage/compute nodes
- Resilience: for availability & the sysadmins' sleep
- Conceptual simplicity to ease onboarding & demistify hosting

The lacking state of the practice

Object storage fitted our needs

- Distributed by design
- Objects are replicated
- Conceptually simple

Existing object stores did not

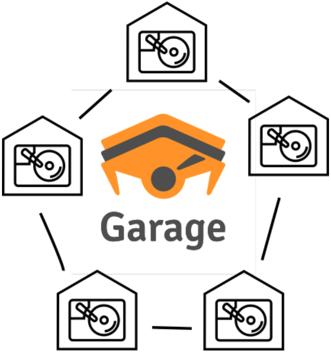
- Too specific / complex
- Resource hungry
- Hidden constraints

We developed Garage, an object store with minimal functionality.
It works, and serves our static sites and media.

Introducing Garage

`garagehq.deuxfleurs.fr`

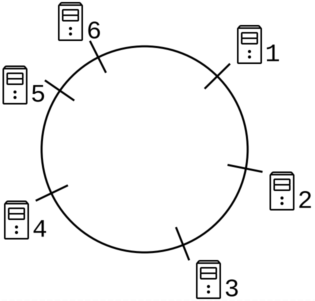
`git.deuxfleurs.fr/Deuxfleurs/garage`



- Distributed data store
- Based on DynamoDB object store (P2P!)
- Modular data types/protocols with CRDTs:
    - Done: objects (media, static sites, backups...) via S3 API
    - To do: e-mails via IMAP protocol, and more

The RING

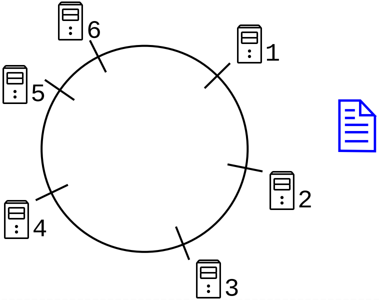
G. DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *ACM SOSP*. New York, USA, 2007



Each node is assigned a unique ID on the circular address space.

The RING

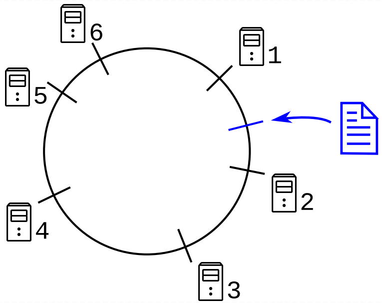
G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *ACM SOSP*. New York, USA, 2007



When a new object is added to the store...

The RING

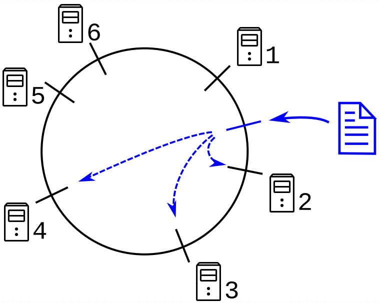
G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *ACM SOSP*. New York, USA, 2007



When a new object is added to the store...
It is assigned a unique ID (its *key*) on the address space.

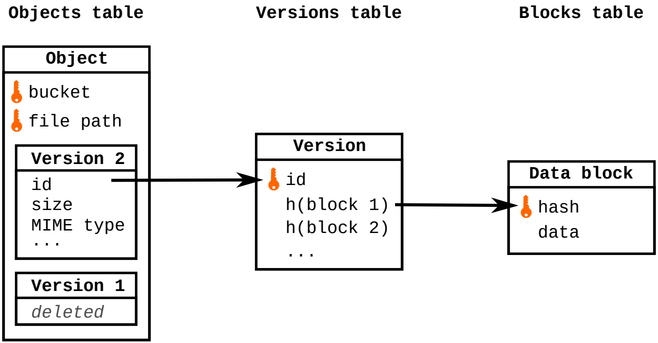
The RING

G. DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *ACM SOSP*. New York, USA, 2007



The R nodes after the object are in charge of replicating it.

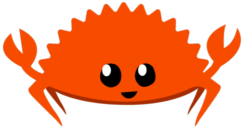
Distributed metadata



The objects, versions and blocks are all stored in the ring.

Written in Rust

Entirely written in Rust!



Pros:

- Compiled and fast
- Features prevent usual mistakes: strongly typed, immutable by default, ownership instead of GC...
- Best of several paradigms: imperative, OO, functional
- Good libraries for network programmings: serialization, http, async/await...

Cons:

- Steep learning curve
- Long compilation times
- Compiler rage

The future is cooler when we bend it our way

Contributions welcome! :D

Thank you for your attention.

Now let's chat!