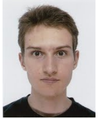
Garage

NGI POINTER

Alex Auvolat, Deuxfleurs Association

`https://garagehq.deuxfleurs.fr/`
Matrix channel: `#garage:deuxfleurs.fr`

Who I am



Alex Auvolat
PhD; co-founder of Deuxfleurs

Deuxfleurs
A non-profit self-hosting collective,
member of the CHATONS network

Our objective at Deuxfleurs

Promote self-hosting and small-scale hosting as an alternative to large cloud providers

Our objective at Deuxfleurs

Promote self-hosting and small-scale hosting as an alternative to large cloud providers

Why is it hard?

Our objective at Deuxfleurs

Promote self-hosting and small-scale hosting
as an alternative to large cloud providers

Why is it hard?

Resilience

(we want good uptime/availability with low supervision)

How to make a stable system

Enterprise-grade systems typically employ:

- RAID
- Redundant power grid + UPS
- Redundant Internet connections
- Low-latency links
- ...

→ it's costly and only worth it at DC scale

How to make a resilient system

Instead, we use:

- Commodity hardware (e.g. old desktop PCs)

How to make a resilient system

How to make a resilient system

How to make a resilient system

Instead, we use:

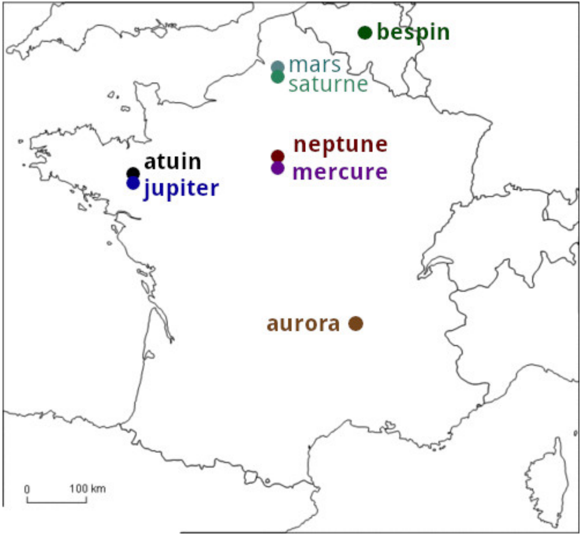
- Commodity hardware (e.g. old desktop PCs)
- Commodity Internet (e.g. FTTB, FTTH) and power grid

How to make a resilient system

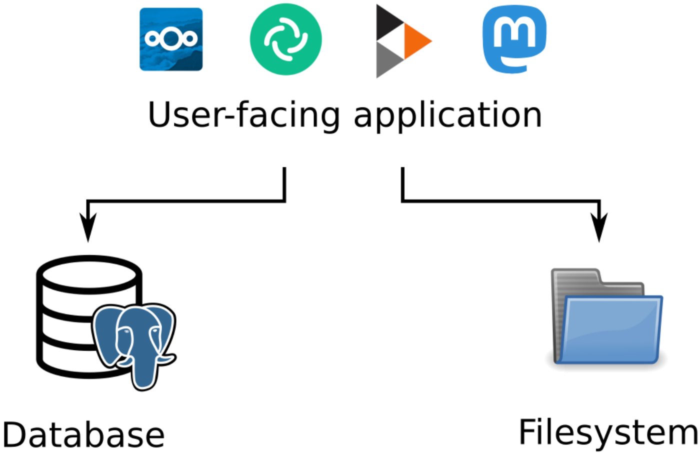
Instead, we use:

- Commodity hardware (e.g. old desktop PCs)
- Commodity Internet (e.g. FTTB, FTTH) and power grid
- **Geographical redundancy** (multi-site replication)

How to make a resilient system



How to make this happen

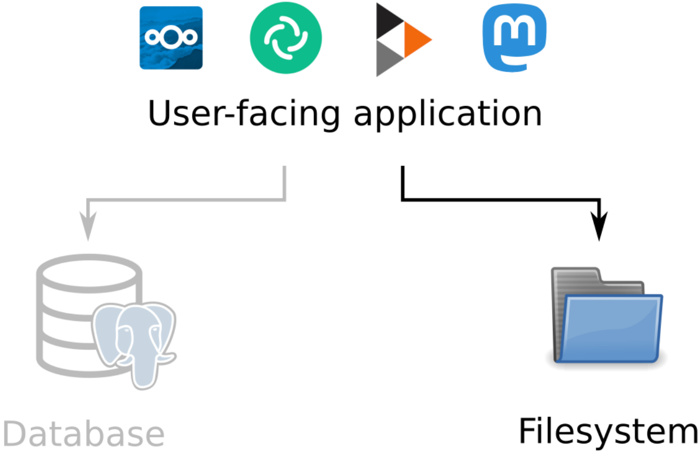


User-facing application

Database

Filesystem

How to make this happen

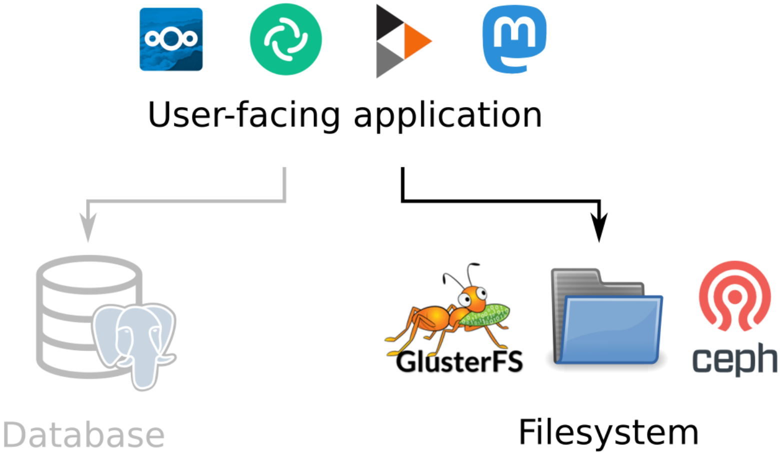


User-facing application

Database

Filesystem

How to make this happen



User-facing application

Database

Filesystem

Distributed file systems are slow

File systems are complex, for example:

- Concurrent modification by several processes
- Folder hierarchies
- Other requirements of the POSIX spec (e.g. locks)

Coordination in a distributed system is costly

Costs explode with commodity hardware / Internet connections
(we experienced this!)

A simpler solution: object storage

Only two operations:

- Put an object at a key
- Retrieve an object from its key

(and a few others)

Sufficient for many applications!

A simpler solution: object storage

S3: a de-facto standard, many compatible applications

MinIO is self-hostable but not suited for geo-distributed deployments

Garage is a self-hosted drop-in replacement for the Amazon S3 object store

The data model of object storage

Object storage is basically a key-value store:

Key: file path + name	**Value: file data + metadata**

`index.html`	`Content-Type: text/html; charset=utf-8`
	`Content-Length: 24929`
	`<binary blob>`

`img/logo.svg`	`Content-Type: text/svg+xml`
	`Content-Length: 13429`
	`<binary blob>`

`download/index.html`	`Content-Type: text/html; charset=utf-8`
	`Content-Length: 26563`
	`<binary blob>`

Two big problems

1. **How to place data on different nodes?**

   Constraints: heterogeneous hardware
   Objective: n copies of everything, maximize usable capacity, maximize resilience

   → the Dynamo model + optimization algorithms

Two big problems

1. **How to place data on different nodes?**

   Constraints: heterogeneous hardware
   Objective: *n* copies of everything, maximize usable capacity, maximize resilience

   → the Dynamo model + optimization algorithms

2. **How to guarantee consistency?**

   Constraints: slow network (geographical distance), node unavailability/crashes
   Objective: maximize availability, read-after-write guarantee

   → CRDTs, monotonicity, read and write quorums

Problem 1: placing data

Key-value stores, upgraded: the Dynamo model

Two keys:

- Partition key: used to divide data into partitions (a.k.a. shards)
- Sort key: used to identify items inside a partition

Partition key: bucket	**Sort key: filename**	**Value**
`website`	`index.html`	(file data)
`website`	`img/logo.svg`	(file data)
`website`	`download/index.html`	(file data)
`backup`	`borg/index.2822`	(file data)
`backup`	`borg/data/2/2329`	(file data)
`backup`	`borg/data/2/2680`	(file data)
`private`	`qq3a2nbe1qjq0ebbvo6ocsp6co`	(file data)

Key-value stores, upgraded: the Dynamo model

- Data with different partition keys is stored independently, on a different set of nodes

  → no easy way to list all partition keys
  → no cross-shard transactions

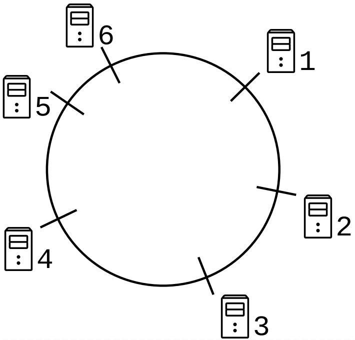
- Placing data: hash the partition key, select nodes accordingly

  → distributed hash table (DHT)

- For a given value of the partition key, items can be listed using their sort keys

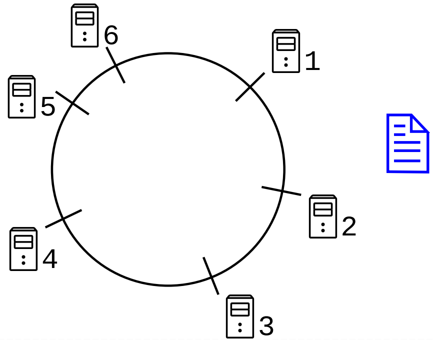
How to spread files over different cluster nodes?

Consistent hashing (Dynamo):



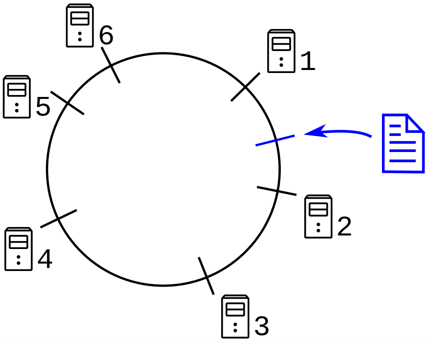
How to spread files over different cluster nodes?

Consistent hashing (Dynamo):



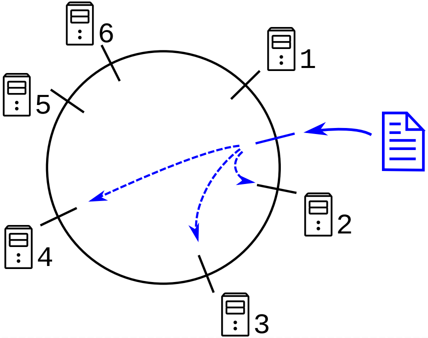
How to spread files over different cluster nodes?

Consistent hashing (Dynamo):

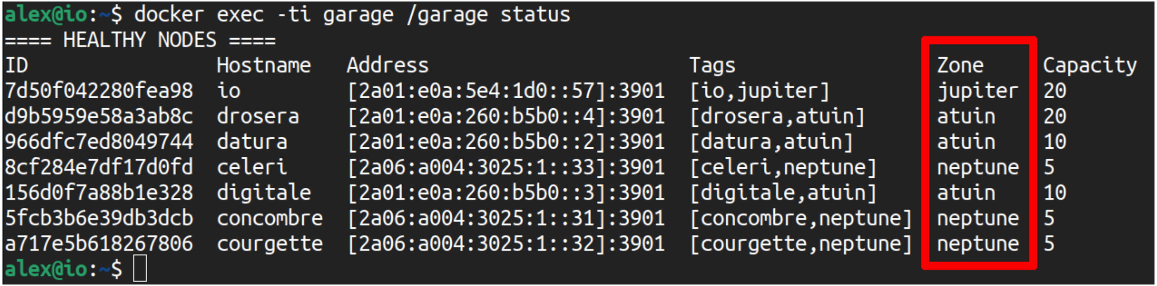


How to spread files over different cluster nodes?

Consistent hashing (Dynamo):



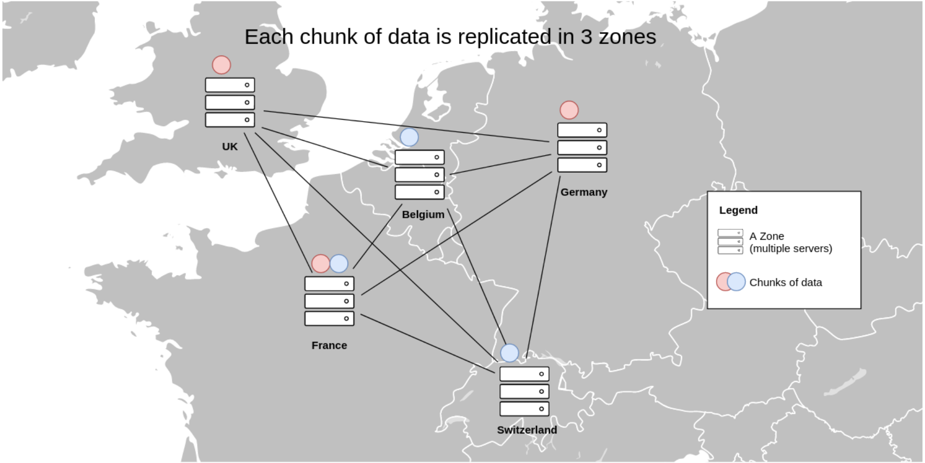
Constraint: location-awareness



```
alex@io:~$ docker exec -ti garage /garage status
==== HEALTHY NODES ====
ID                Hostname   Address                      Tags                     Zone     Capacity
7d50f042280fea98  io         [2a01:e0a:5e4:1d0::57]:3901  [io,jupiter]             jupiter  20
d9b5959e58a3ab8c  drosera    [2a01:e0a:260:b5b0::4]:3901  [drosera,atuin]          atuin    20
966dfc7ed8049744  datura     [2a01:e0a:260:b5b0::2]:3901  [datura,atuin]           atuin    10
8cf284e7df17d0fd  celeri     [2a06:a004:3025:1::33]:3901  [celeri,neptune]         neptune  5
156d0f7a88b1e328  digitale   [2a01:e0a:260:b5b0::3]:3901  [digitale,atuin]         atuin    10
5fcb3b6e39db3dcb  concombre  [2a06:a004:3025:1::31]:3901  [concombre,neptune]      neptune  5
a717e5b618267806  courgette  [2a06:a004:3025:1::32]:3901  [courgette,neptune]      neptune  5
alex@io:~$
```

Garage replicates data on different zones when possible

Constraint: location-awareness



Each chunk of data is replicated in 3 zones

Issues with consistent hashing

- Consistent hashing doesn't dispatch data based on geographical location of nodes

Issues with consistent hashing

- Consistent hashing doesn't dispatch data based on geographical location of nodes
- Geographically aware adaptation, try 1:
  data quantities not well balanced between nodes

Issues with consistent hashing

- Consistent hashing doesn't dispatch data based on geographical location of nodes
- Geographically aware adaptation, try 1:
  data quantities not well balanced between nodes
- Geographically aware adaptation, try 2:
  too many reshuffles when adding/removing nodes

How to spread files over different cluster nodes?

Garage's method: build an index table

Realization: we can actually precompute an optimal solution

How to spread files over different cluster nodes?

Garage's method: build an index table

Realization: we can actually precompute an optimal solution

Partition	**Node 1**	**Node 2**	**Node 3**
Partition 0	Io (jupiter)	Drosera (atuin)	Courgette (neptune)
Partition 1	Datura (atuin)	Courgette (neptune)	Io (jupiter)
Partition 2	Io(jupiter)	Celeri (neptune)	Drosera (atuin)
⋮	⋮	⋮	⋮
Partition 255	Concombre (neptune)	Io (jupiter)	Drosera (atuin)

How to spread files over different cluster nodes?

Garage's method: build an index table

Realization: we can actually precompute an optimal solution

Partition	**Node 1**	**Node 2**	**Node 3**
Partition 0	Io (jupiter)	Drosera (atuin)	Courgette (neptune)
Partition 1	Datura (atuin)	Courgette (neptune)	Io (jupiter)
Partition 2	Io(jupiter)	Celeri (neptune)	Drosera (atuin)
⋮	⋮	⋮	⋮
Partition 255	Concombre (neptune)	Io (jupiter)	Drosera (atuin)

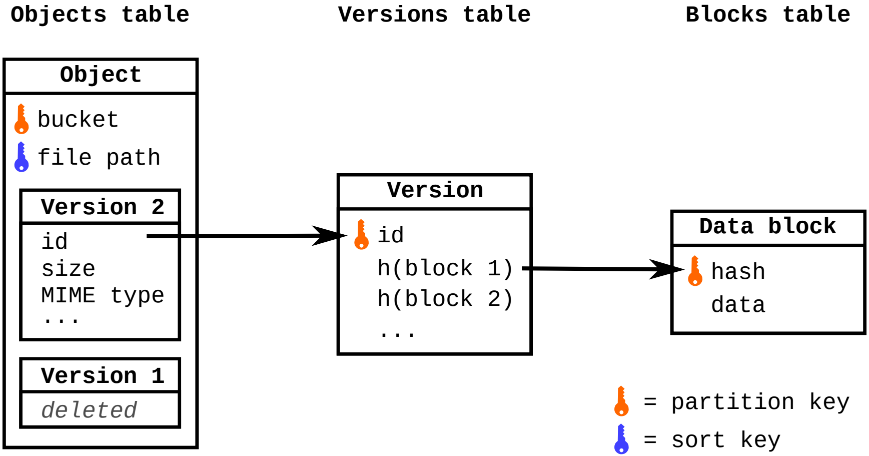
The index table is built centrally using an optimal algorithm,
then propagated to all nodes

The relationship between *partition* and *partition key*

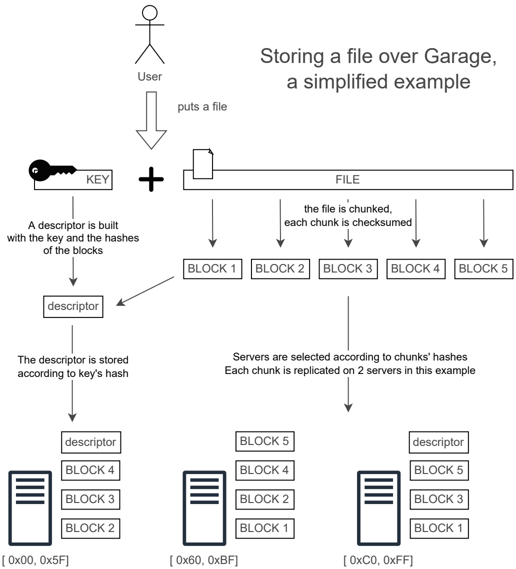
Partition key	Partition	Sort key	Value
`website`	Partition 12	`index.html`	(file data)
`website`	Partition 12	`img/logo.svg`	(file data)
`website`	Partition 12	`download/index.html`	(file data)
`backup`	Partition 42	`borg/index.2822`	(file data)
`backup`	Partition 42	`borg/data/2/2329`	(file data)
`backup`	Partition 42	`borg/data/2/2680`	(file data)
`private`	Partition 42	`qq3a2nbe1qjq0ebbvo6ocsp6co`	(file data)

To read or write an item: hash partition key
→ determine partition number (first 8 bits)
→ find associated nodes

Garage's internal data structures



Storing and retrieving files



Storing a file over Garage,
a simplified example

User

puts a file

KEY

FILE

A descriptor is built
with the key and the hashes
of the blocks

the file is chunked,
each chunk is checksumed

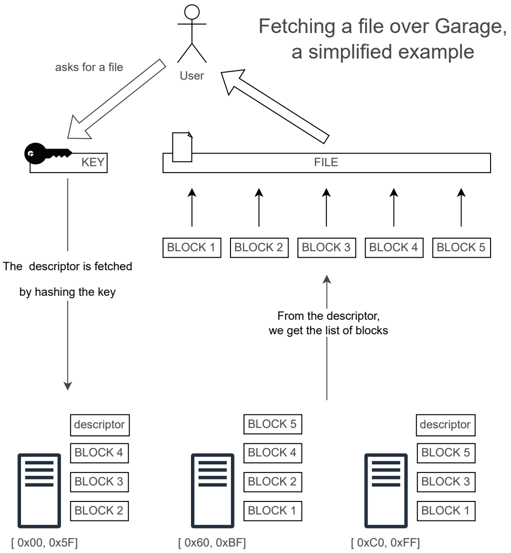
BLOCK 1 BLOCK 2 BLOCK 3 BLOCK 4 BLOCK 5

descriptor

The descriptor is stored
according to key's hash

Servers are selected according to chunks' hashes
Each chunk is replicated on 2 servers in this example

Storing and retrieving files



Fetching a file over Garage,
a simplified example

Problem 2: ensuring consistency

Consensus vs weak consistency

Consensus-based systems:

- **Leader-based:** a leader is elected to coordinate all reads and writes
- **Linearizability** of all operations (strongest consistency guarantee)
- Any sequential specification can be implemented as a **replicated state machine**
- **Costly**, the leader is a bottleneck; leader elections on failure take time

Consensus vs weak consistency

Consensus-based systems:

- **Leader-based:** a leader is elected to coordinate all reads and writes
- **Linearizability** of all operations (strongest consistency guarantee)
- Any sequential specification can be implemented as a **replicated state machine**
- **Costly**, the leader is a bottleneck; leader elections on failure take time

Weakly consistent systems:

- **Nodes are equivalent**, any node can originate a read or write operation
- **Read-after-write consistency** with quorums, eventual consistency without
- **Operations have to commute**, i.e. we can only implement CRDTs
- **Fast**, no single bottleneck; works the same with offline nodes

Consensus vs weak consistency

From a theoretical point of view:

Consensus-based systems:

Require **additional assumptions** such as a fault detector or a strong RNG (FLP impossibility theorem)

Weakly consistent systems:

Can be implemented in **any asynchronous message passing distributed system** with node crashes

They represent **different classes of computational capability**

Consensus vs weak consistency

The same objects cannot be implemented in both models.

Consensus-based systems:

Any sequential specification

Easier to program for: just write your program as if it were sequential on a single machine

Weakly consistent systems:

Only CRDTs
(conflict-free replicated data types)

Part of the complexity is **reported to the consumer of the API**

Understanding the power of consensus

Consensus: an API with a single operation, *propose*(x)

1. nodes all call *propose*(x) with their proposed value;
2. nodes all receive the same value as a return value, which is one of the proposed values

Understanding the power of consensus

Consensus: an API with a single operation, $propose(x)$

1. nodes all call $propose(x)$ with their proposed value;
2. nodes all receive the same value as a return value, which is one of the proposed values

Equivalent to a distributed algorithm that gives a total order on all requests

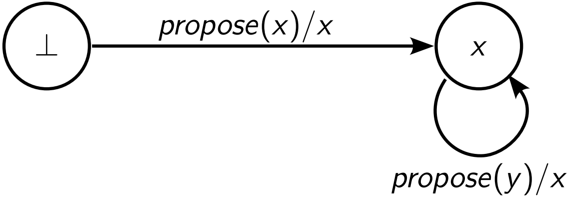
Understanding the power of consensus

Consensus: an API with a single operation, *propose*(x)

1. nodes all call *propose*(x) with their proposed value;
2. nodes all receive the same value as a return value, which is one of the proposed values

Equivalent to a distributed algorithm that gives a total order on all requests

Implemented by this simple replicated state machine:



Can my object be implemented without consensus?

Given the specification of an API:

- **Using this API, we can implement the consensus object** (the *propose* function)
  → the API is equivalent to consensus/total ordering of messages
  → the API cannot be implemented in a weakly consistent system

Can my object be implemented without consensus?

Given the specification of an API:

- **Using this API, we can implement the consensus object** (the *propose* function)
  → the API is equivalent to consensus/total ordering of messages
  → the API cannot be implemented in a weakly consistent system

- **This API can be implemented using only weak primitives**
  (e.g. in the asynchronous message passing model with no further assumption)
  → the API is strictly weaker than consensus
  → we can implement it in Garage!

Why avoid consensus?

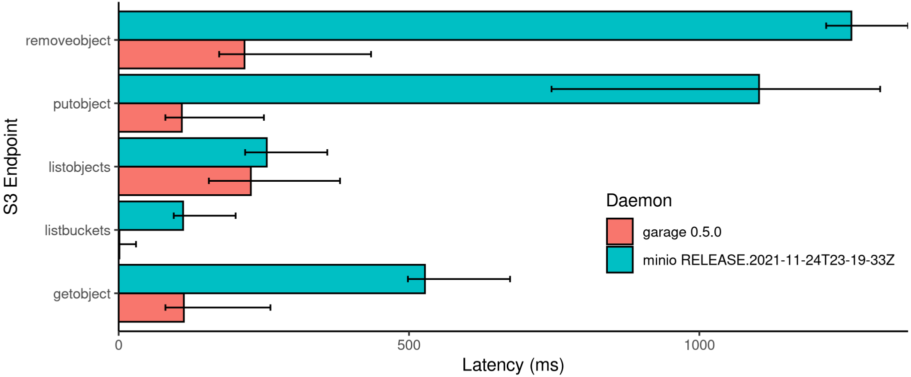
Consensus can be implemented reasonably well in practice, so why avoid it?

- **Software complexity:** RAFT and PAXOS are complex beasts;
  harder to prove, harder to reason about
- **Performance issues:**
  - Theoretical requirements (RNG, failure detector) translate into **practical costs**
  - The leader is a **bottleneck** for all requests;
    even in leaderless approaches, **all nodes must process all operations in order**
  - Particularly **sensitive to higher latency** between nodes

Performance gains in practice

S3 endpoint latency in a simulated geo-distributed cluster

100 measurements, 6 nodes in 3 DC (2 nodes/DC), 100ms RTT + 20ms jitter between DC
no contention: latency is due to intra-cluster communications
colored bar = mean latency, error bar = min and max latency



Get the code to reproduce this graph at https://git.deuxfleurs.fr/quentin/benchmarks

What can we implement without consensus?

- Any **conflict-free replicated data type** (CRDT)

What can we implement without consensus?

- Any **conflict-free replicated data type** (CRDT)
- Non-transactional key-value stores such as S3 are equivalent to a simple CRDT: a map of **last-writer-wins registers** (each key is its own CRDT)

What can we implement without consensus?

- Any **conflict-free replicated data type** (CRDT)
- Non-transactional key-value stores such as S3 are equivalent to a simple CRDT:
  a map of **last-writer-wins registers** (each key is its own CRDT)
- **Read-after-write consistency** can be implemented using quorums on read and write operations

What can we implement without consensus?

- Any **conflict-free replicated data type** (CRDT)
- Non-transactional key-value stores such as S3 are equivalent to a simple CRDT: a map of **last-writer-wins registers** (each key is its own CRDT)
- **Read-after-write consistency** can be implemented using quorums on read and write operations
- **Monotonicity of reads** can be implemented with repair-on-read (makes reads more costly, not implemented in Garage)

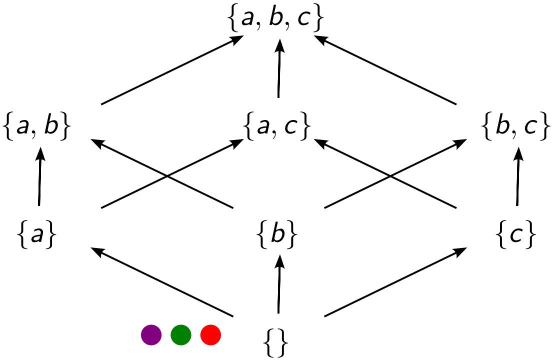
CRDTs and quorums: read-after-write consistency

$$\begin{tikzcd}
 & \{a, b, c\} & \\
\{a, b\} \arrow[ur] & \{a, c\} \arrow[u] & \{b, c\} \arrow[ul] \\
\{a\} \arrow[u] \arrow[ur] & \{b\} \arrow[ul] \arrow[ur] & \{c\} \arrow[ul] \arrow[u] \\
 & \{\} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

CRDTs and quorums: read-after-write consistency

$write(\{a\})$:

- ● $\not\supseteq \{a\}$
- ● $\not\supseteq \{a\}$
- ● $\not\supseteq \{a\}$



$\{a, b, c\}$

$\{a, b\}$ $\{a, c\}$ $\{b, c\}$

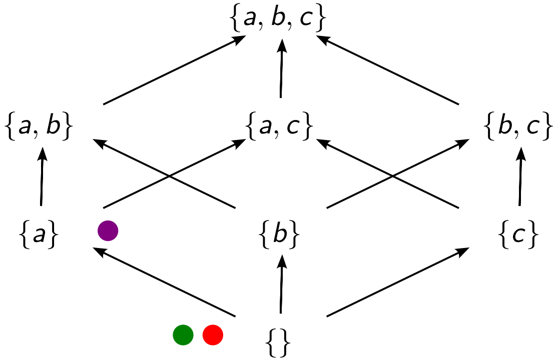
$\{a\}$ $\{b\}$ $\{c\}$

● ● ● $\{\}$

CRDTs and quorums: read-after-write consistency

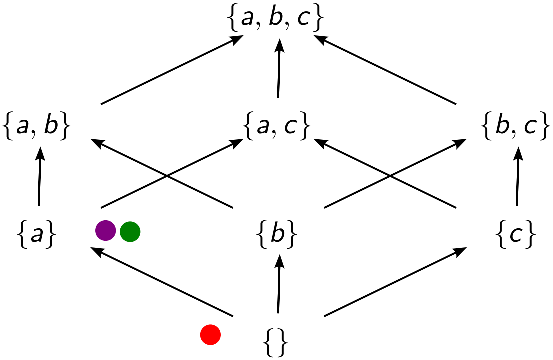
$write(\{a\})$:

- ● $\sqsupseteq \{a\} \rightarrow$ OK
- ● $\not\sqsupseteq \{a\}$
- ● $\not\sqsupseteq \{a\}$



CRDTs and quorums: read-after-write consistency

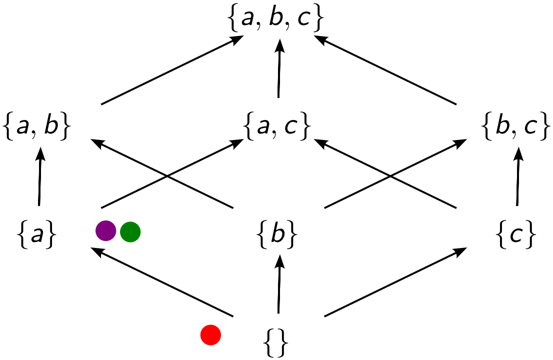
```
$write(\{a\})$:
    ● $\sqsupseteq \{a\} \rightarrow$ OK
    ● $\sqsupseteq \{a\} \rightarrow$ OK
    ● $\not\sqsupseteq \{a\}$
    return OK
```



CRDTs and quorums: read-after-write consistency

```
$write(\{a\})$:
    ● $\sqsupseteq \{a\} \rightarrow$ OK
    ● $\sqsupseteq \{a\} \rightarrow$ OK
    ● $\not\sqsupseteq \{a\}$
    return OK

$read()$:
```



CRDTs and quorums: read-after-write consistency

write({*a*}):

● $\sqsupseteq \{a\}$ → OK

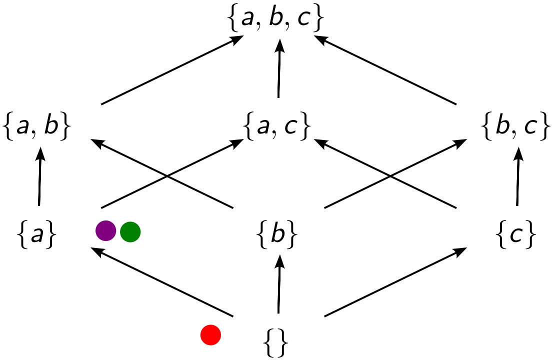
● $\sqsupseteq \{a\}$ → OK

● $\not\sqsupseteq \{a\}$

return OK

read():

● → {}



CRDTs and quorums: read-after-write consistency

$\textit{write}(\{a\})$:

🟣 $\sqsupseteq \{a\} \rightarrow$ OK

🟢 $\sqsupseteq \{a\} \rightarrow$ OK

🔴 $\not\sqsupseteq \{a\}$

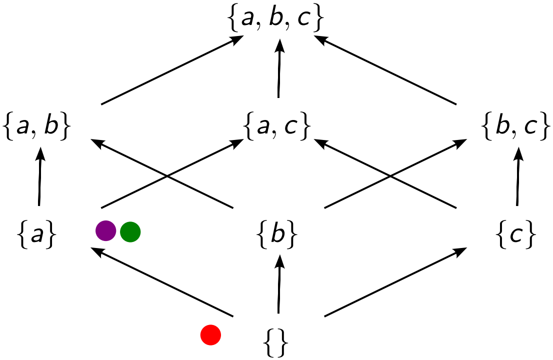
return OK

$\textit{read}()$:

🔴 $\rightarrow \{\}$

🟣 $\rightarrow \{a\}$

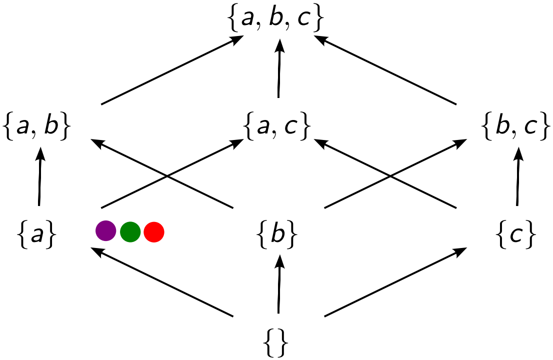
return $\{\} \sqcup \{a\} = \{a\}$



CRDTs and quorums: read-after-write consistency

```
$write(\{a\})$:
    🟣 $\sqsupseteq \{a\} \rightarrow$ OK
    🟢 $\sqsupseteq \{a\} \rightarrow$ OK
    🔴 $\sqsupseteq \{a\}$
    return OK

$read()$:
    🔴 $\rightarrow \{\}$
    🟣 $\rightarrow \{a\}$
    return $\{\} \sqcup \{a\} = \{a\}$
```



CRDTs and quorums: read-after-write consistency

Property: If node A did an operation $\textit{write}(x)$ and received an OK response,
and node B starts an operation $\textit{read}()$ after A received OK,
then B will read a value $x' \sqsupseteq x$.

Algorithm $\textit{write}(x)$:

1. Broadcast $\textit{write}(x)$ to all nodes
2. Wait for $k > n/2$ nodes to reply OK
3. Return OK

Algorithm $\textit{read}()$:

1. Broadcast $\textit{read}()$ to all nodes
2. Wait for $k > n/2$ nodes to reply with values $x_1, \ldots, x_k$
3. Return $x_1 \sqcup \ldots \sqcup x_k$

Why does it work? There is at least one node at the intersection between the two sets of nodes that replied to each request, that “saw” x before the $\textit{read}()$ started ($x_i \sqsupseteq x$).

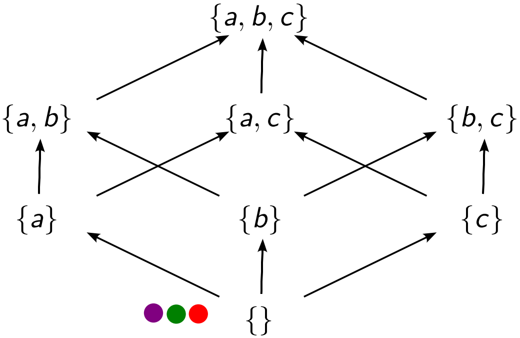
CRDTs and quorums: monotonic-reads consistency

$write(\{a\})$:

- ● $\not\supseteq \{a\}$
- ● $\not\supseteq \{a\}$
- ● $\not\supseteq \{a\}$

$write(\{b\})$:

- ● $\not\supseteq \{b\}$
- ● $\not\supseteq \{b\}$
- ● $\not\supseteq \{b\}$



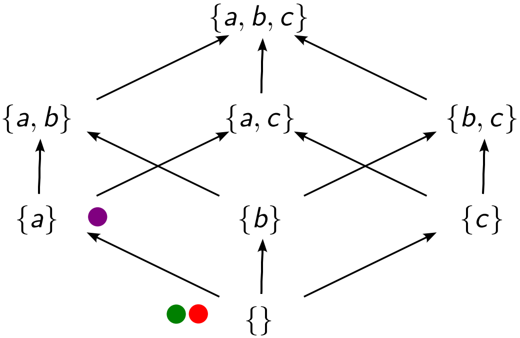
CRDTs and quorums: monotonic-reads consistency

write$(\{a\})$:

- ● $\sqsupseteq \{a\} \rightarrow$ OK
- ● $\not\sqsupseteq \{a\}$
- ● $\not\sqsupseteq \{a\}$

write$(\{b\})$:

- ● $\not\sqsupseteq \{b\}$
- ● $\not\sqsupseteq \{b\}$
- ● $\not\sqsupseteq \{b\}$



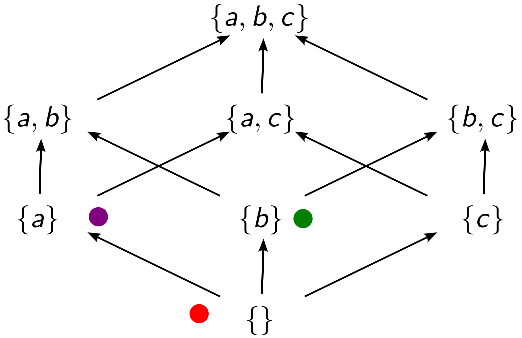
CRDTs and quorums: monotonic-reads consistency

write$(\{a\})$:

- ● $\sqsupseteq \{a\} \rightarrow$ OK
- ● $\not\sqsupseteq \{a\}$
- ● $\not\sqsupseteq \{a\}$

write$(\{b\})$:

- ● $\not\sqsupseteq \{b\}$
- ● $\sqsupseteq \{b\} \rightarrow$ OK
- ● $\not\sqsupseteq \{b\}$



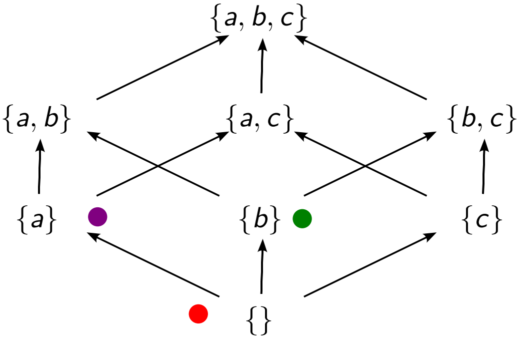
CRDTs and quorums: monotonic-reads consistency

write($\{a\}$):

- 🟣 $\sqsupseteq \{a\} \rightarrow$ OK
- 🟢 $\not\sqsupseteq \{a\}$
- 🔴 $\not\sqsupseteq \{a\}$

write($\{b\}$):

- 🟣 $\not\sqsupseteq \{b\}$
- 🟢 $\sqsupseteq \{b\} \rightarrow$ OK
- 🔴 $\not\sqsupseteq \{b\}$



read():

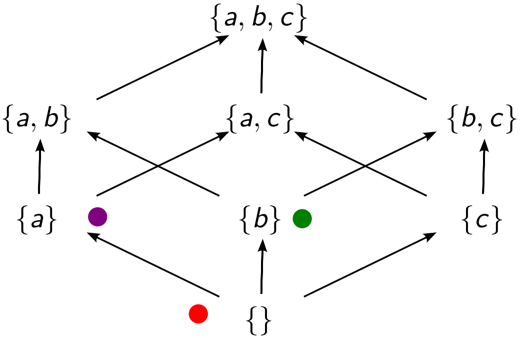
CRDTs and quorums: monotonic-reads consistency

write($\{a\}$):

- 🟣 $\sqsupseteq \{a\} \rightarrow$ OK
- 🟢 $\not\sqsupseteq \{a\}$
- 🔴 $\not\sqsupseteq \{a\}$

write($\{b\}$):

- 🟣 $\not\sqsupseteq \{b\}$
- 🟢 $\sqsupseteq \{b\} \rightarrow$ OK
- 🔴 $\not\sqsupseteq \{b\}$



read():

- 🟣 $\rightarrow \{a\}$

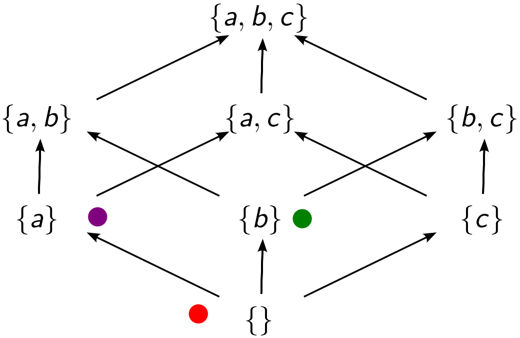
CRDTs and quorums: monotonic-reads consistency

$write(\{a\})$:

- 🟣 $\sqsupseteq \{a\} \to \text{OK}$
- 🟢 $\not\sqsupseteq \{a\}$
- 🔴 $\not\sqsupseteq \{a\}$

$write(\{b\})$:

- 🟣 $\not\sqsupseteq \{b\}$
- 🟢 $\sqsupseteq \{b\} \to \text{OK}$
- 🔴 $\not\sqsupseteq \{b\}$



$read()$:

- 🟣 $\to \{a\}$
- 🔴 $\to \{\}$

return $\{a\}$

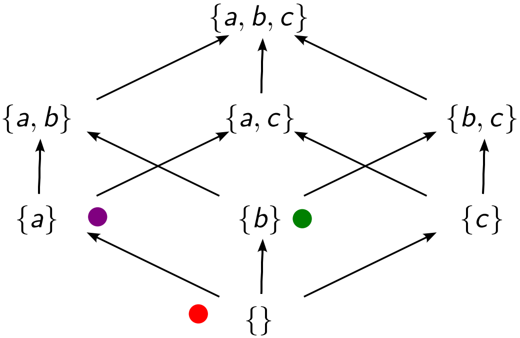
CRDTs and quorums: monotonic-reads consistency

$write(\{a\})$:

🟣 $\sqsupseteq \{a\} \rightarrow$ OK
🟢 $\not\sqsupseteq \{a\}$
🔴 $\not\sqsupseteq \{a\}$

$write(\{b\})$:

🟣 $\not\sqsupseteq \{b\}$
🟢 $\sqsupseteq \{b\} \rightarrow$ OK
🔴 $\not\sqsupseteq \{b\}$



$read()$:

🟣 $\rightarrow \{a\}$
🔴 $\rightarrow \{\}$
return $\{a\}$

;

$read()$:

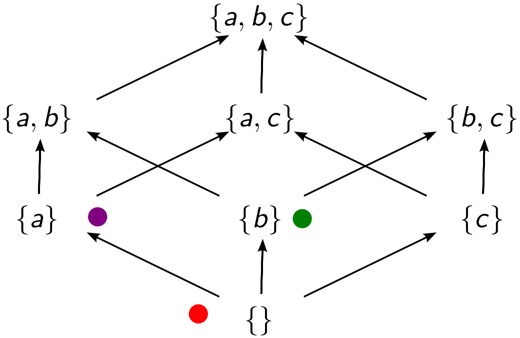
CRDTs and quorums: monotonic-reads consistency

write$(\{a\})$:

🟣 $\sqsupseteq \{a\} \to$ OK
🟢 $\not\sqsupseteq \{a\}$
🔴 $\not\sqsupseteq \{a\}$

write$(\{b\})$:

🟣 $\not\sqsupseteq \{b\}$
🟢 $\sqsupseteq \{b\} \to$ OK
🔴 $\not\sqsupseteq \{b\}$



read():

🟣 $\to \{a\}$
🔴 $\to \{\}$
return $\{a\}$

;

read():

🔴 $\to \{\}$

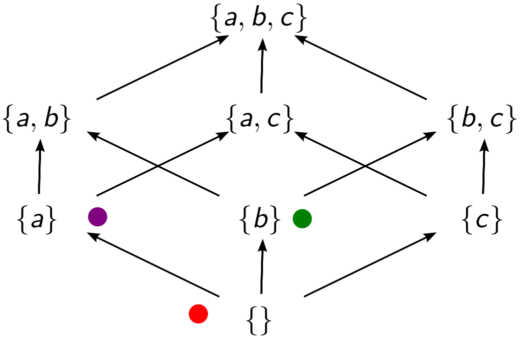
CRDTs and quorums: monotonic-reads consistency

$write(\{a\})$:

- ● $\sqsupseteq \{a\} \rightarrow$ OK
- ● $\not\sqsupseteq \{a\}$
- ● $\not\sqsupseteq \{a\}$

$write(\{b\})$:

- ● $\not\sqsupseteq \{b\}$
- ● $\sqsupseteq \{b\} \rightarrow$ OK
- ● $\not\sqsupseteq \{b\}$



$read()$:

- ● $\rightarrow \{a\}$
- ● $\rightarrow \{\}$

return $\{a\}$

;

$read()$:

- ● $\rightarrow \{\}$
- ● $\rightarrow \{b\}$

return $\{b\}$

CRDTs and quorums: monotonic-reads consistency

$write(\{a\})$:

🟣 $\sqsupseteq \{a\} \rightarrow$ OK

🟢 $\not\sqsupseteq \{a\}$

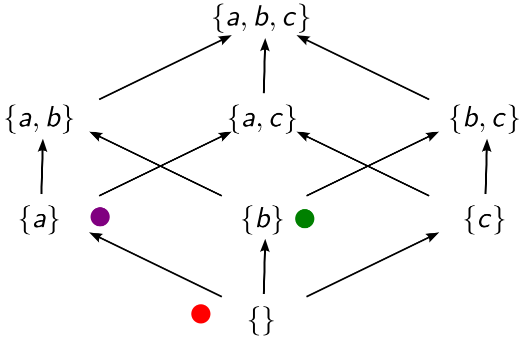
🔴 $\not\sqsupseteq \{a\}$

$write(\{b\})$:

🟣 $\not\sqsupseteq \{b\}$

🟢 $\sqsupseteq \{b\} \rightarrow$ OK

🔴 $\not\sqsupseteq \{b\}$



$read()$:

🟣 $\rightarrow \{a\}$

🔴 $\rightarrow \{\}$

return $\{a\}$

;

$read()$:

🔴 $\rightarrow \{\}$

🟢 $\rightarrow \{b\}$

return $\{b\}$

??!

$\{a\} \not\sqsubseteq \{b\}$

CRDTs and quorums: monotonic-reads consistency

Property: If node A did an operation *read*() and received x as a response,
and node B starts an operation *read*() after A received x,
then B will read a value $x' \sqsupseteq x$.

Algorithm *monotonic_read*()**:** (a.k.a. repair-on-read)

1. Broadcast *read*() to all nodes
2. Wait for $k > n/2$ nodes to reply with values $x_1, \dots, x_k$
3. If $x_i \neq x_j$ for some nodes i and j,
   then call $\mathit{write}(x_1 \sqcup \dots \sqcup x_k)$ and wait for OK from $k' > n/2$ nodes
4. Return $x_1 \sqcup \dots \sqcup x_k$

This makes reads slower in some cases, and is **not implemented in Garage**.

A hard problem: layout changes

- We rely on quorums $k > n/2$ within each partition:

$$n = 3, \qquad k \geq 2$$

A hard problem: layout changes

- We rely on quorums $k > n/2$ within each partition:

$$n = 3, \qquad k \geq 2$$

- When rebalancing, the set of nodes responsible for a partition can change:

$$\{n_A, n_B, n_C\} \rightarrow \{n_A, n_D, n_E\}$$

A hard problem: layout changes

- We rely on quorums $k > n/2$ within each partition:

$$n = 3, \qquad k \geq 2$$

- When rebalancing, the set of nodes responsible for a partition can change:

$$\{n_A, n_B, n_C\} \rightarrow \{n_A, n_D, n_E\}$$

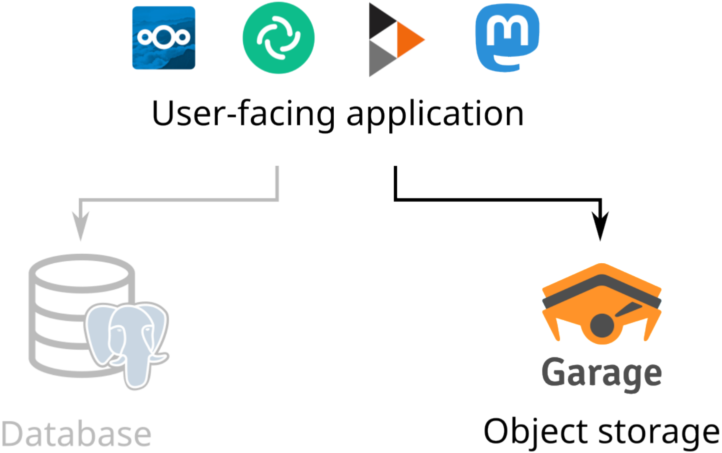
- During the rebalancing, D and E don't yet have the data,
  and B and C want to get rid of the data to free up space

  → quorums only within the new set of nodes don't work

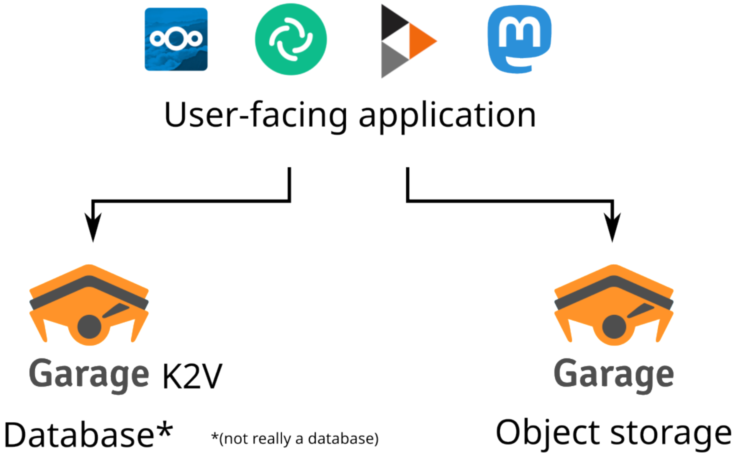
  → how to coordinate? **currently, we don't...**

Going further than the S3 API

Using Garage for everything



Using Garage for everything



User-facing application

Garage K2V

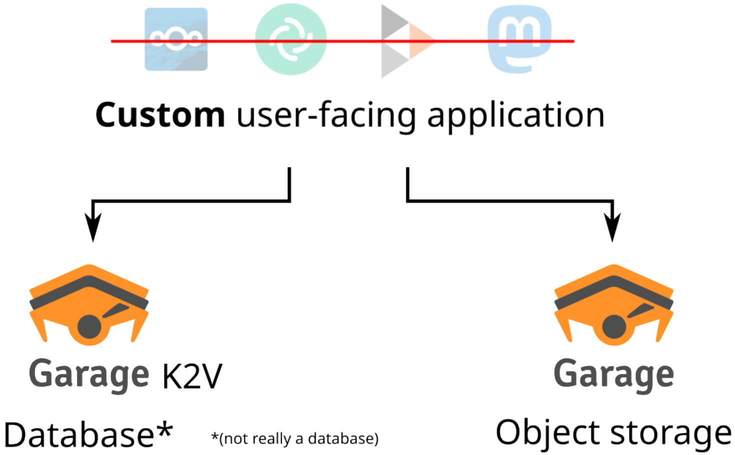
Database*

*(not really a database)

Garage

Object storage

Using Garage for everything



Custom user-facing application

Garage K2V

Database*

*(not really a database)

Garage

Object storage

K2V Design

- A new, custom, minimal API
  - Single-item operations
  - Operations on ranges and batches of items
  - Polling operations to help implement a PubSub pattern

K2V Design

- A new, custom, minimal API
  - Single-item operations
  - Operations on ranges and batches of items
  - Polling operations to help implement a PubSub pattern

- Exposes the partitoning mechanism of Garage
  K2V = partition key / sort key / value (like Dynamo)

K2V Design

- A new, custom, minimal API
  - Single-item operations
  - Operations on ranges and batches of items
  - Polling operations to help implement a PubSub pattern
- Exposes the partitoning mechanism of Garage
  K2V = partition key / sort key / value (like Dynamo)
- Weakly consistent, CRDT-friendly
  → no support for transactions (not ACID)

K2V Design

- A new, custom, minimal API
  - Single-item operations
  - Operations on ranges and batches of items
  - Polling operations to help implement a PubSub pattern
- Exposes the partitoning mechanism of Garage
  K2V = partition key / sort key / value (like Dynamo)
- Weakly consistent, CRDT-friendly
  → no support for transactions (not ACID)
- Cryptography-friendly: values are binary blobs

Handling concurrent values

How to handle concurrency? Example:

1. Client A reads the initial value of a key, x_0

Handling concurrent values

How to handle concurrency? Example:

1. Client A reads the initial value of a key, x_0
2. Client B also reads the initial value x_0 of that key

Handling concurrent values

How to handle concurrency? Example:

1. Client A reads the initial value of a key, x_0
2. Client B also reads the initial value x_0 of that key
3. Client A modifies x_0, and writes a new value x_1

Handling concurrent values

How to handle concurrency? Example:

1. Client A reads the initial value of a key, x_0
2. Client B also reads the initial value x_0 of that key
3. Client A modifies x_0, and writes a new value x_1
4. Client B also modifies x_0, and writes a new value x'_1, without having a chance to first read x_1

   → what should the final state be?

Handling concurrent values

- If we keep only x_1 or x'_1, we risk **loosing application data**

Handling concurrent values

- If we keep only x_1 or x'_1, we risk **loosing application data**
- Values are opaque binary blobs, **K2V cannot resolve conflicts** by itself (e.g. by implementing a CRDT)

Handling concurrent values

- If we keep only x_1 or x'_1, we risk **loosing application data**
- Values are opaque binary blobs, **K2V cannot resolve conflicts** by itself (e.g. by implementing a CRDT)
- Solution: **we keep both!**
  $\rightarrow$ the value of the key is now $\{x_1, x'_1\}$
  $\rightarrow$ the client application can decide how to resolve conflicts on the next read

Keeping track of causality

How does K2V know that x_1 and x_1' are concurrent?

- *read*() returns **a set of values** and an associated **causality token**

Keeping track of causality

How does K2V know that x_1 and x_1' are concurrent?

- *read*() returns **a set of values** and an associated **causality token**
- When calling *write*(), the client sends **the causality token from its last read**

Keeping track of causality

How does K2V know that x_1 and x'_1 are concurrent?

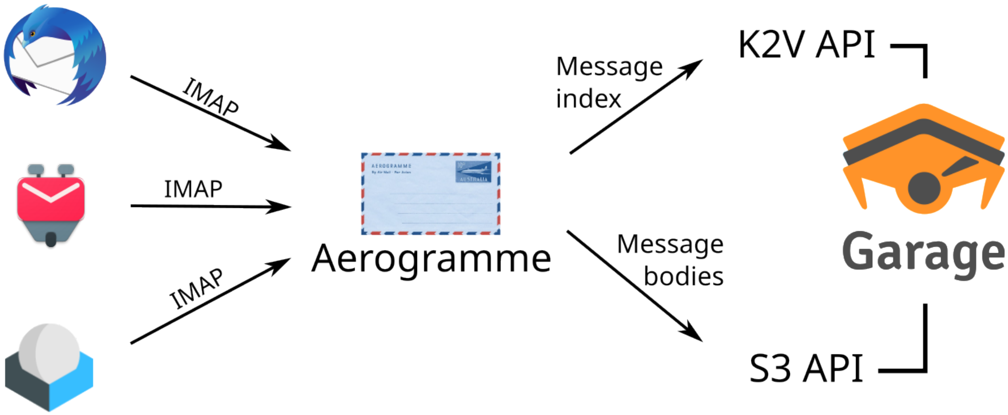
- *read*() returns **a set of values** and an associated **causality token**
- When calling *write*(), the client sends **the causality token from its last read**
- The causality token represents the set of values **already seen by the client**
  → those values are the **causal past** of the write operation
  → K2V can keep concurrent values and overwrite all ones in the causal past

Keeping track of causality

How does K2V know that x_1 and x'_1 are concurrent?

- *read*() returns **a set of values** and an associated **causality token**
- When calling *write*(), the client sends **the causality token from its last read**
- The causality token represents the set of values **already seen by the client**
  → those values are the **causal past** of the write operation
  → K2V can keep concurrent values and overwrite all ones in the causal past
- Internally, the causality token is **a vector clock**

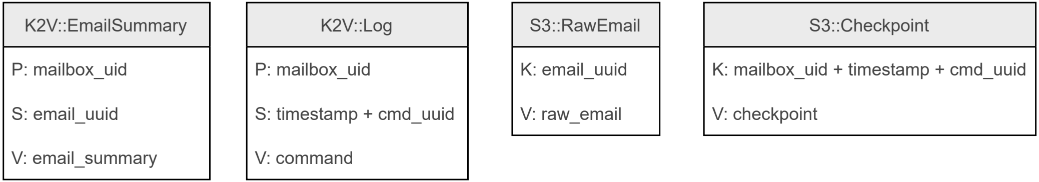
Application: an e-mail storage server



Aerogramme data model

	immutable	mutable
K2V	Email Summary	Log
S3	Full Email	Checkpoint

Aerogramme data model



K2V::EmailSummary

P: mailbox_uid

S: email_uuid

V: email_summary

K2V::Log

P: mailbox_uid

S: timestamp + cmd_uuid

V: command

S3::RawEmail

K: email_uuid

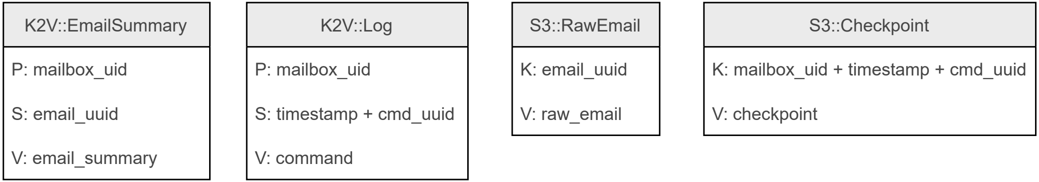
V: raw_email

S3::Checkpoint

K: mailbox_uid + timestamp + cmd_uuid

V: checkpoint

Aerogramme data model



K2V::EmailSummary

P: mailbox_uid

S: email_uuid

V: email_summary

K2V::Log

P: mailbox_uid

S: timestamp + cmd_uuid

V: command

S3::RawEmail

K: email_uuid

V: raw_email

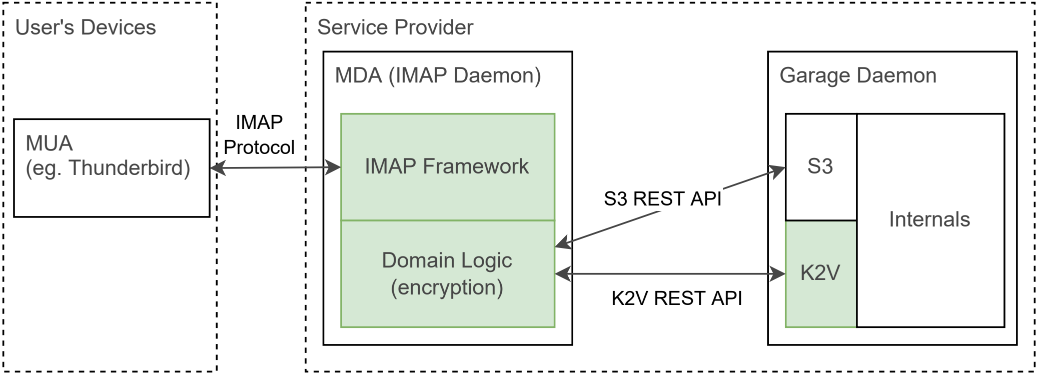
S3::Checkpoint

K: mailbox_uid + timestamp + cmd_uuid

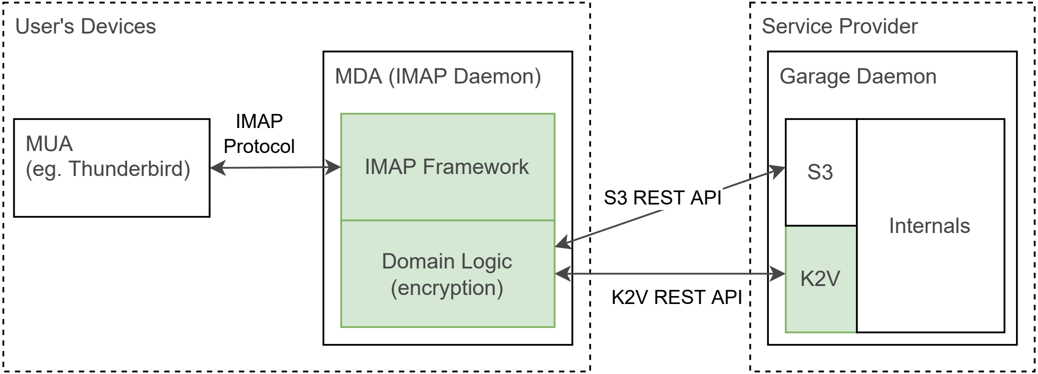
V: checkpoint

Aerogramme encrypts all stored values for privacy
(Garage server administrators can't read your mail)

Different deployment scenarios



Different deployment scenarios



A new model for building resilient software

How to build an application using only Garage as a data store:

1. Design a data model suited to K2V
   (see Cassandra docs on porting SQL data models to Cassandra)

   - Use CRDTs or other eventually consistent data types (see e.g. Bayou)
   - Store opaque binary blobs to provide End-to-End Encryption

A new model for building resilient software

How to build an application using only Garage as a data store:

1. Design a data model suited to K2V
   (see Cassandra docs on porting SQL data models to Cassandra)

   - Use CRDTs or other eventually consistent data types (see e.g. Bayou)

   - Store opaque binary blobs to provide End-to-End Encryption

2. Store big blobs (files) using the S3 API

A new model for building resilient software

How to build an application using only Garage as a data store:

1. Design a data model suited to K2V
   (see Cassandra docs on porting SQL data models to Cassandra)

   - Use CRDTs or other eventually consistent data types (see e.g. Bayou)
   - Store opaque binary blobs to provide End-to-End Encryption

2. Store big blobs (files) using the S3 API

3. Let Garage manage sharding, replication, failover, etc.

Conclusion

Perspectives

- Fix the consistency issue when rebalancing
- Write about Garage's architecture and properties,
  and about our proposed architecture for (E2EE) apps over K2V+S3
- Continue developing Garage; finish Aerogramme; build new applications...
- Anything else?

Where to find us

https://garagehq.deuxfleurs.fr/
mailto:garagehq@deuxfleurs.fr
#garage:deuxfleurs.fr on Matrix