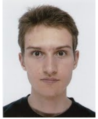
Garage

Deuxfleurs Association

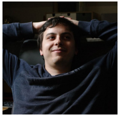
https://garagehq.deuxfleurs.fr/
Matrix channel: #garage:deuxfleurs.fr

Who we are



Alex Auvolat
PhD; co-founder of Deuxfleurs

Quentin Dufour
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?

<u>Resilience</u>

(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:

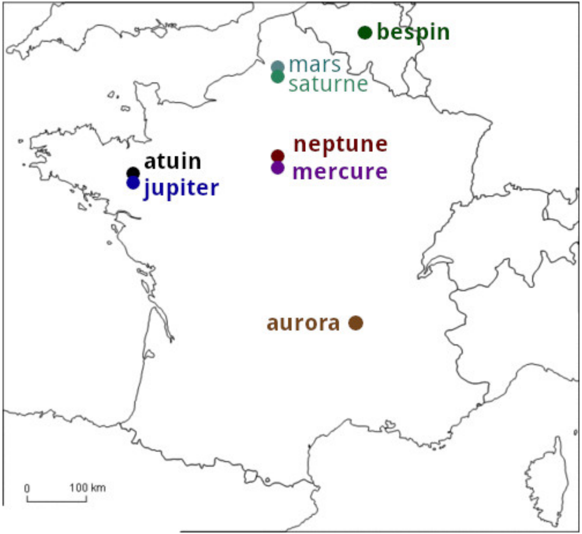
- Commodity hardware (e.g. old desktop PCs)
- Commodity Internet (e.g. FTTB, FTTH) and power grid

How to make a resilient system

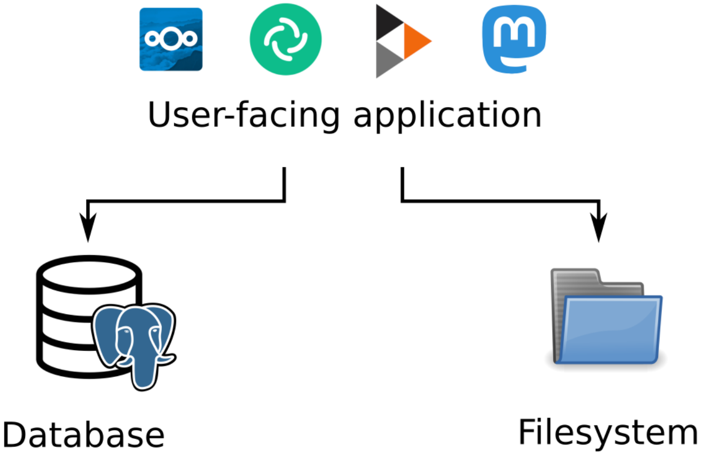
Instead, we use:

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

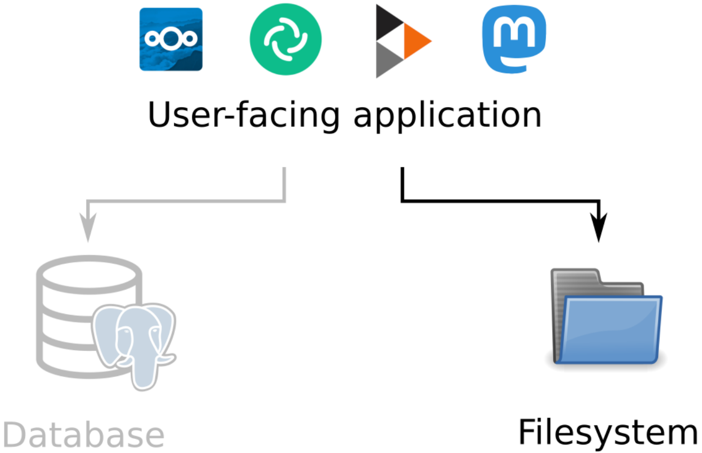
How to make a resilient system



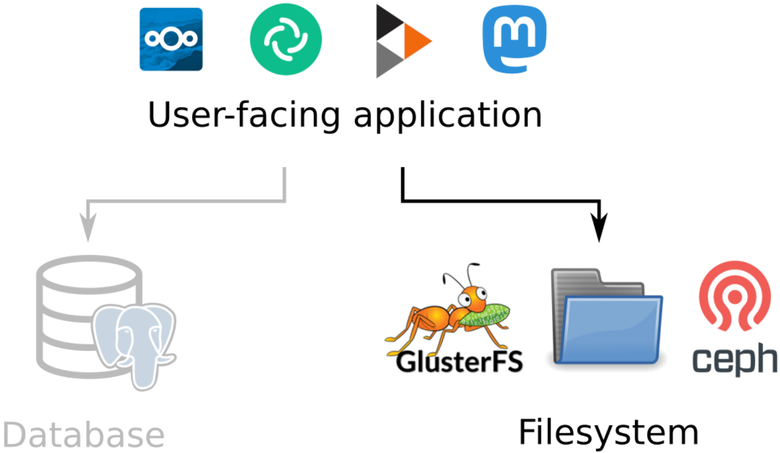
How to make this happen



How to make this happen



How to make this happen



Distributed file systems are slow

File systems are complex, for example:

- Concurrent modification by several processes
- Folder hierarchies
- Other requirements of the POSIX spec

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 (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 independantly, on a different set of nodes

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

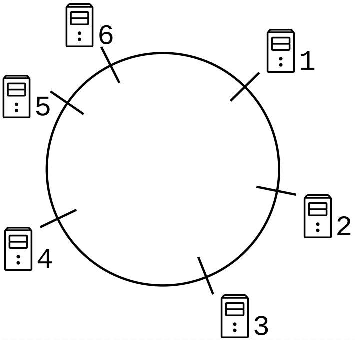
- Placing data: hash the partition key, select nodes accordingly

  → distributed hash table (DHT)

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

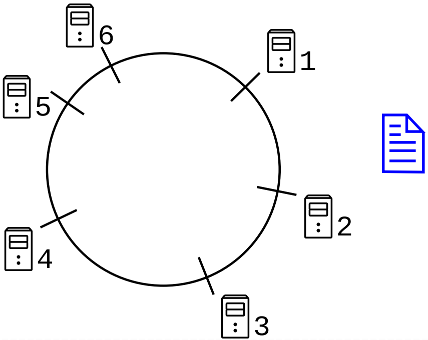
How to spread files over different cluster nodes?

Consistent hashing (Dynamo):



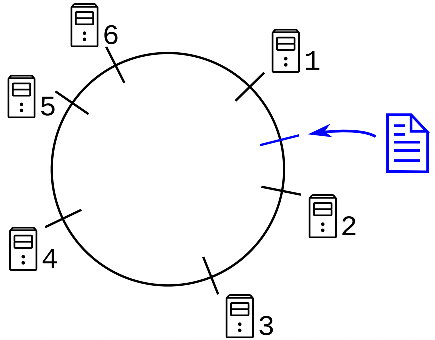
How to spread files over different cluster nodes?

Consistent hashing (Dynamo):



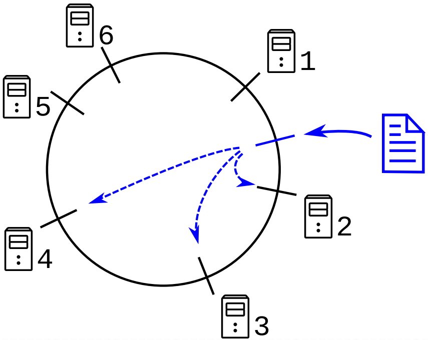
How to spread files over different cluster nodes?

Consistent hashing (Dynamo):



How to spread files over different cluster nodes?

Consistent hashing (Dynamo):

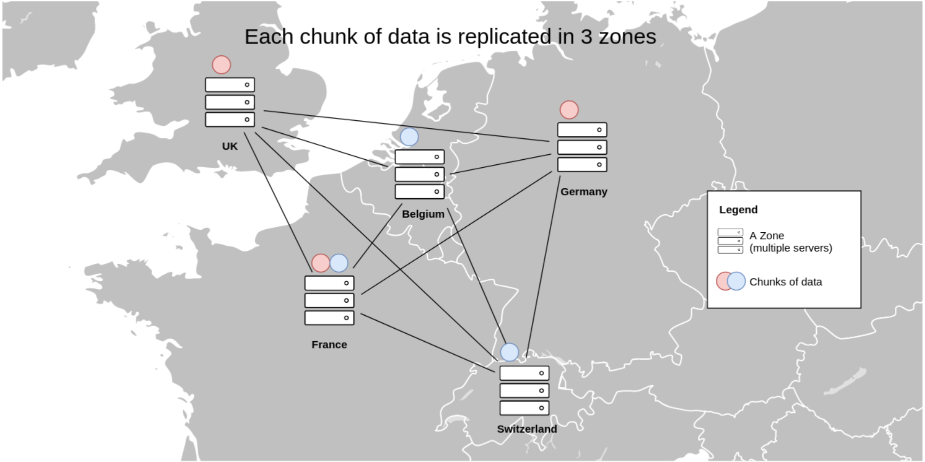


Constraint: location-awareness

```
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



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)

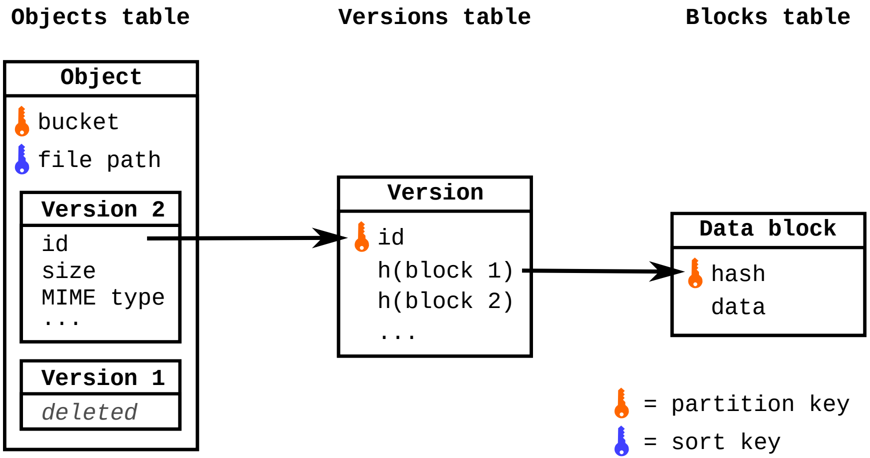
The index table is built centrally using an optimal algorithm,
then propagated to all nodes

The relationship between *partition* and *partition key*

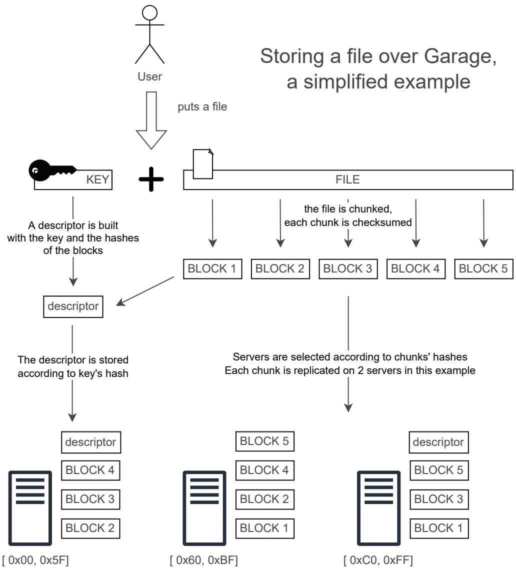
Partition key	**Partition**	**Sort key**	**Value**
`website`	Partition 12	`index.html`	(file data)
`website`	Partition 12	`img/logo.svg`	(file data)
`website`	Partition 12	`download/index.html`	(file data)
`backup`	Partition 42	`borg/index.2822`	(file data)
`backup`	Partition 42	`borg/data/2/2329`	(file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

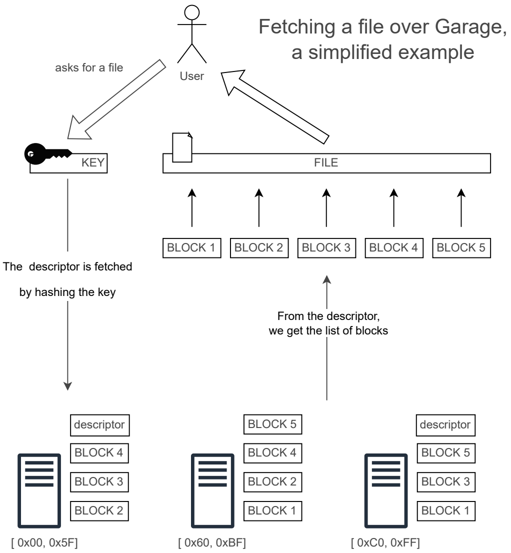
Garage's internal data structures



Storing and retrieving files



Storing and retrieving files



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

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:

CRDTs only
(conflict-free replicated data types)

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

Consensus vs weak consistency

From a theoretical point of view:

Consensus-based systems:

Require **additionnal assumptions** such as a fault detector or a strong RNG

Weakly consistent systems:

Can be implemented in **any asynchronous message passing distributed system**

They represent **different classes of computational capability**

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

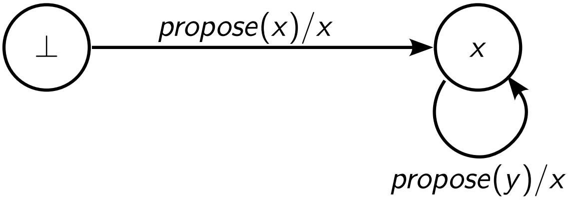
Understanding the power of consensus

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

- **This API can be implemented using only weak primitives**
  (e.g. a bunch of atomic registers)
  → the API is strictly weaker than consensus
  → we can implement it in Garage!

Why avoid consensus?

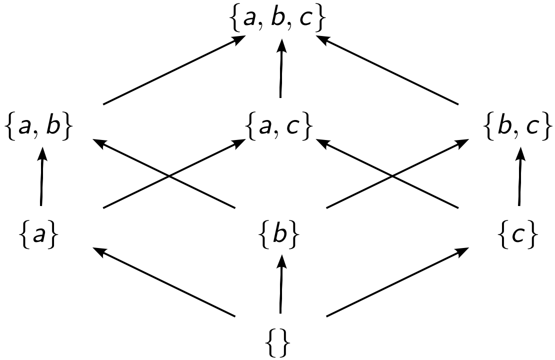
Consensus can be implemented reasonably well in practice, so why avoid it?

- **Software complexity:** RAFT and PAXOS are complex beasts; harder to prove, harder to reason about
- **Performance issues:**
  - The leader is a **bottleneck** for all requests
  - Particularly **sensitive to higher latency** between nodes

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 **last-writer-wins registry**
- **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)

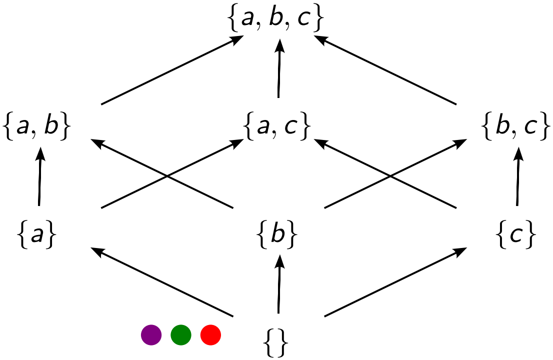
Understanding CRDTs and quorums



Understanding CRDTs and quorums

$write(\{a\})$:

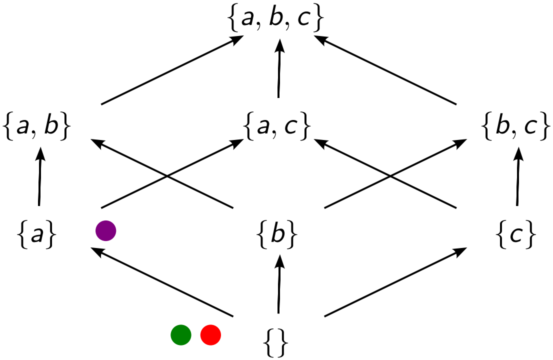
- ● $\not\supseteq \{a\}$
- ● $\not\supseteq \{a\}$
- ● $\not\supseteq \{a\}$



Understanding CRDTs and quorums

$write(\{a\})$:

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

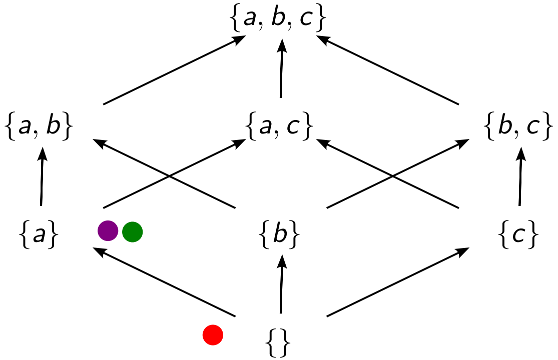


Understanding CRDTs and quorums

write($\{a\}$):

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

return OK

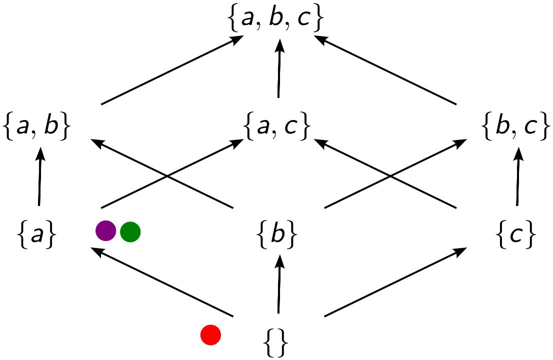


Understanding CRDTs and quorums

$write(\{a\})$:

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

$read()$:



Understanding CRDTs and quorums

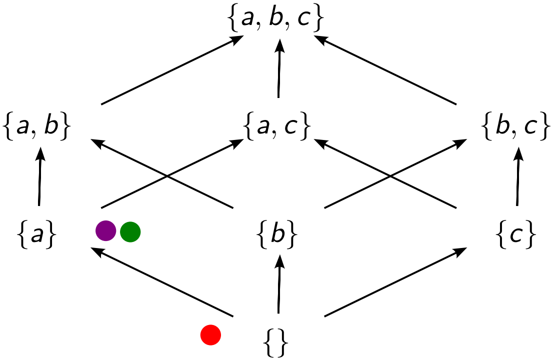
write$(\{a\})$:

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

return OK

read():

- ● $\rightarrow \{\}$



Understanding CRDTs and quorums

write$(\{a\})$:

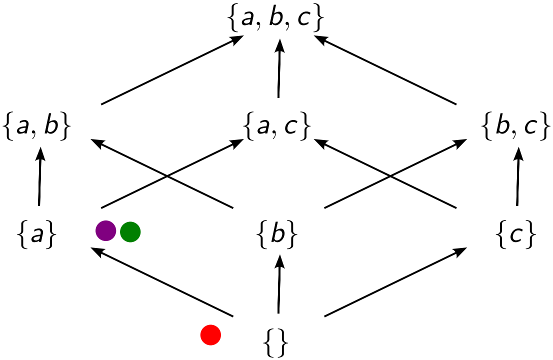
- 🟣 $\sqsupseteq \{a\} \rightarrow$ OK
- 🟢 $\sqsupseteq \{a\} \rightarrow$ OK
- 🔴 $\not\sqsupseteq \{a\}$

return OK

read$()$:

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

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



Understanding CRDTs and quorums

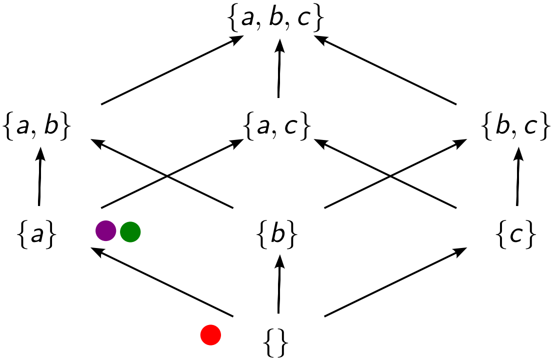
$write(\{a\})$:

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

return OK

$write(\{b\})$:

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



Understanding CRDTs and quorums

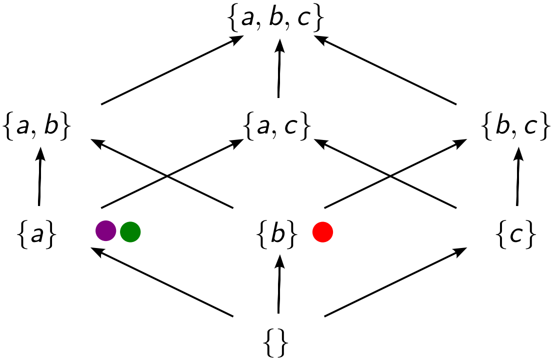
$write(\{a\})$:

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

return OK

$write(\{b\})$:

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



Understanding CRDTs and quorums

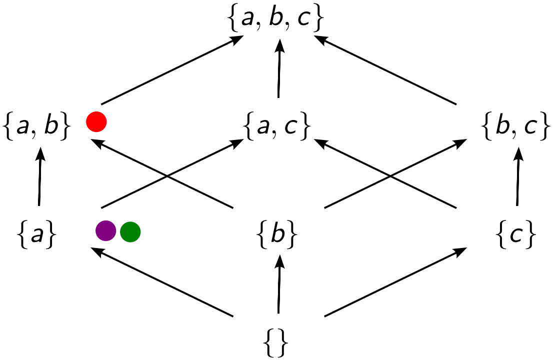
$write(\{a\})$:

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

return OK

$write(\{b\})$:

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



Understanding CRDTs and quorums

$write(\{a\})$:

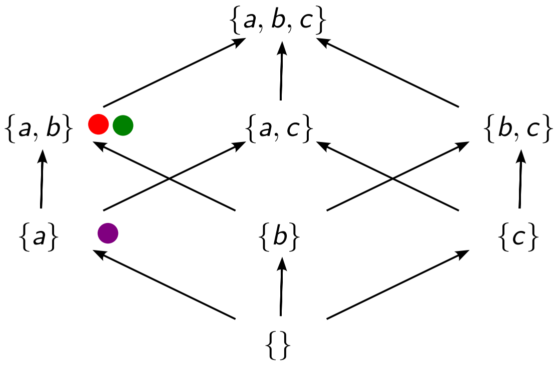
- 🟣 $\sqsupseteq \{a\} \rightarrow$ OK
- 🟢 $\sqsupseteq \{a\} \rightarrow$ OK
- 🔴 $\sqsupseteq \{a\}$

return OK

$write(\{b\})$:

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

return OK



Understanding CRDTs and quorums

$write(\{a\})$:

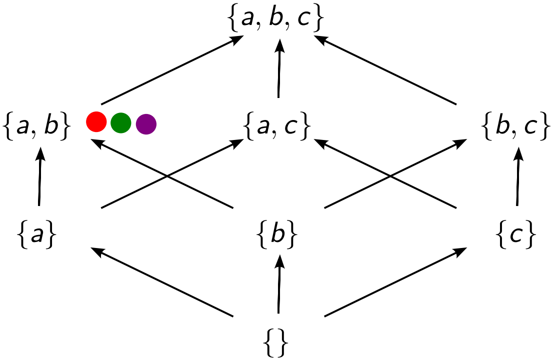
- ● $\sqsupseteq \{a\} \rightarrow$ OK
- ● $\sqsupseteq \{a\} \rightarrow$ OK
- ● $\sqsupseteq \{a\}$

return OK

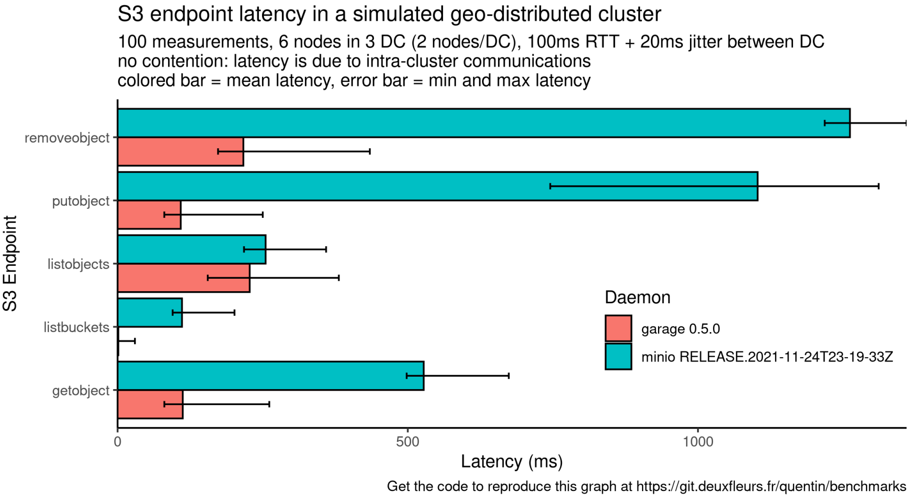
$write(\{b\})$:

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

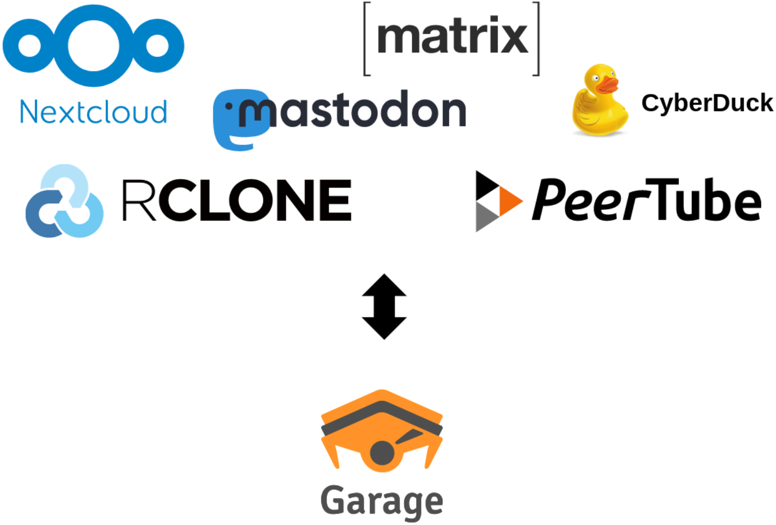
return OK



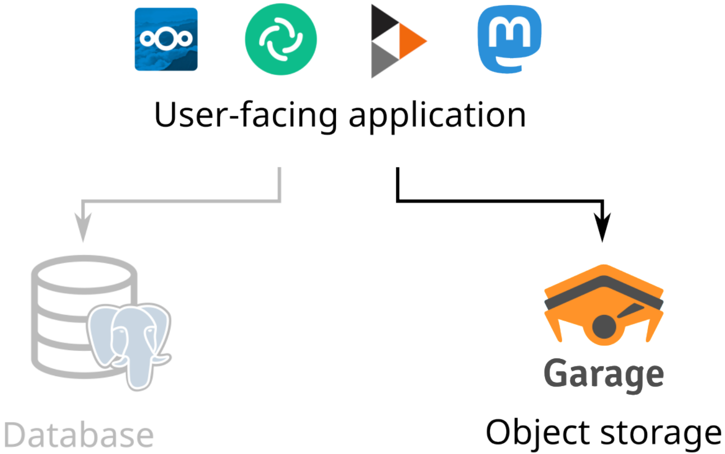
Performance gains in practice



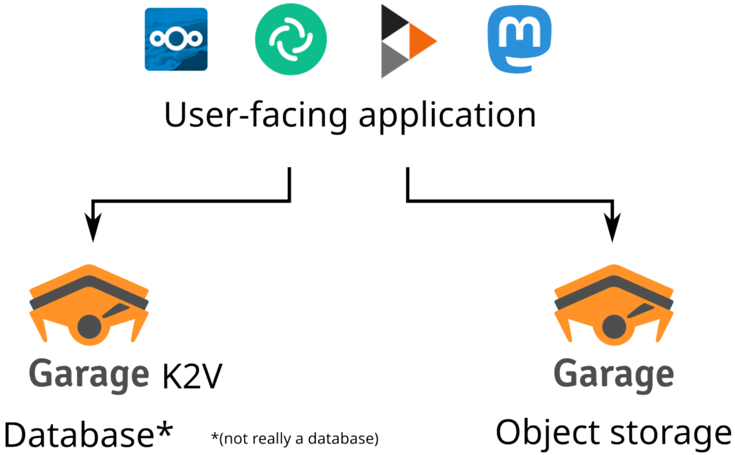
An ever-increasing compatibility list



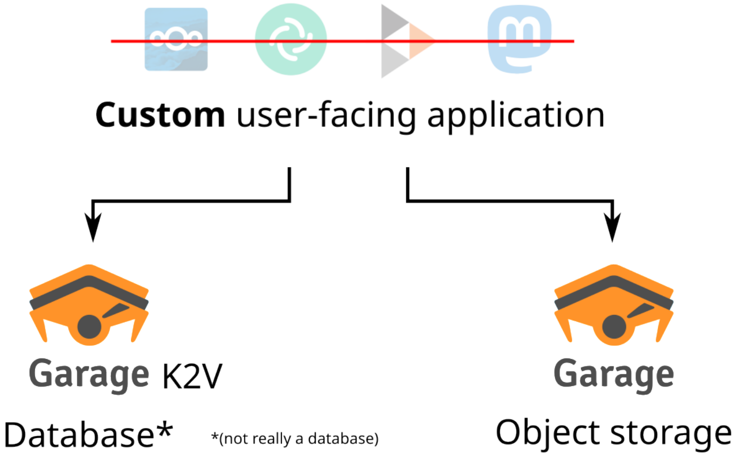
Further plans for Garage



Further plans for Garage



Further plans for Garage



K2V Design

- A new, custom, minimal API

K2V Design

- A new, custom, minimal API
- Exposes the partitoning mechanism of Garage
  K2V = partition key / sort key / value (like Dynamo)

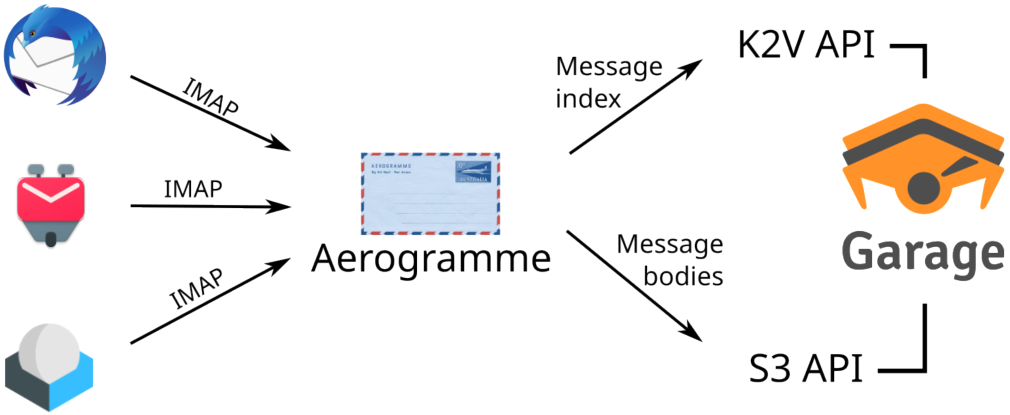
K2V Design

- A new, custom, minimal API
- Exposes the partitoning mechanism of Garage
  K2V = partition key / sort key / value (like Dynamo)
- Coordination-free, CRDT-friendly (inspired by Riak)

K2V Design

- A new, custom, minimal API
- Exposes the partitoning mechanism of Garage
  K2V = partition key / sort key / value (like Dynamo)
- Coordination-free, CRDT-friendly (inspired by Riak)
- Cryptography-friendly: values are binary blobs

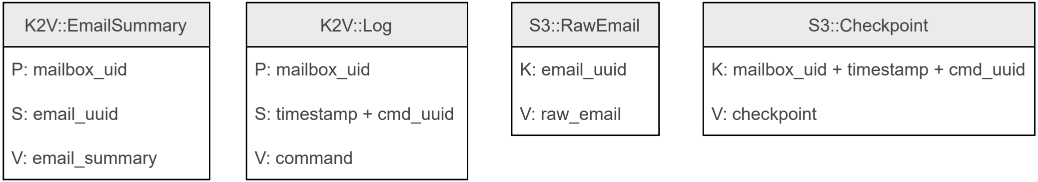
Application: an e-mail storage server



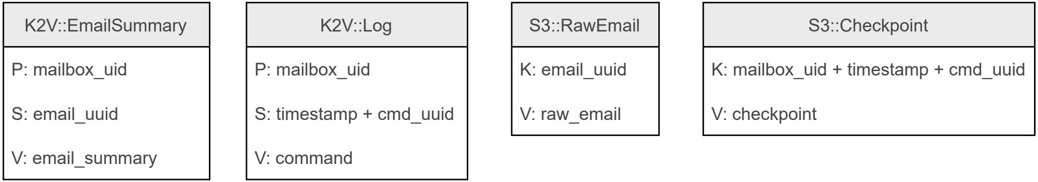
Aerogramme data model

	immutable	mutable
K2V	Email Summary	Log
S3	Full Email	Checkpoint

Aerogramme data model

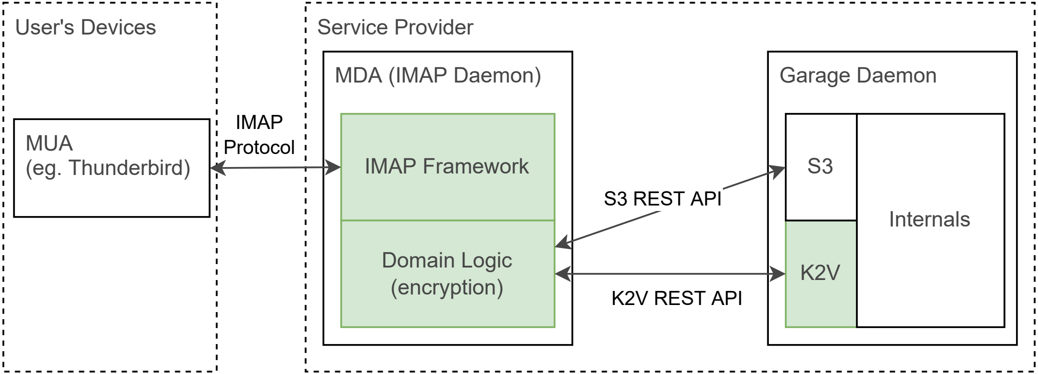


Aerogramme data model

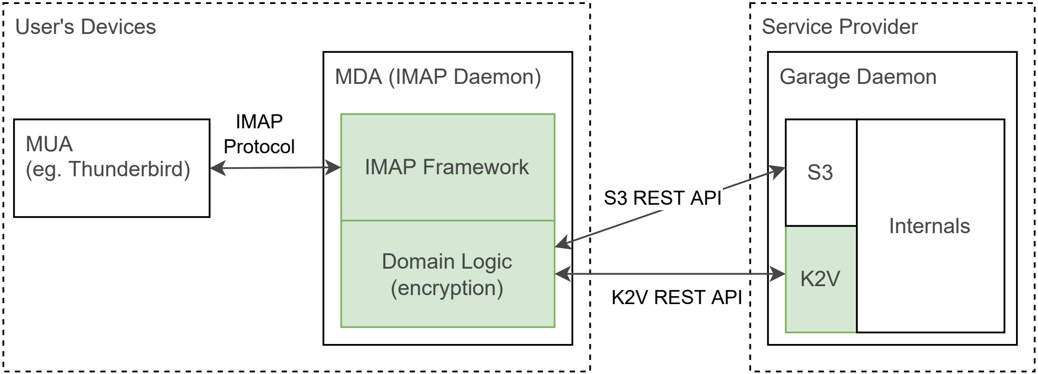


Aerogramme encrypts all stored values for privacy
(Garage server administrators can't read your mail)

Different deployment scenarios



Different deployment scenarios



A new model for building resilient software

- 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
- Store big blobs (files) in S3
- Let Garage manage sharding, replication, failover, etc.

Research perspectives

- Write about Garage's global architecture *(paper in progress)*
- Measure and improve Garage's performances
- Discuss the optimal layout algorithm, provide proofs
- Write about our proposed architecture for (E2EE) apps over K2V+S3

Where to find us

Garage

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

AGPLv3 Free Software
Free as in Freedom