Causality for the Cloudlets: Offering Causality on the Edge With Small Metadata

Nuno Afonso, Manuel Bravo, Luís Rodrigues
Processing on the edge

There are many mobile applications that require the execution of resource demanding tasks.

• Face recognition
• Video-indexing
• Augmented reality
Processing on the edge

There are many mobile applications that require the execution of resource demanding tasks.

- Face recognition
- Video-indexing
- Augmented reality

These tasks need to be processed in the cloud.
Processing on the edge

There are many mobile applications that require the execution of resource demanding tasks.

- Face recognition
- Video-indexing
- Augmented reality

**Latency constraints: 5-30 ms !!**
Processing on the edge

There are many mobile applications that require the execution of resource demanding tasks.

- Face recognition
- Video-indexing
- Augmented reality

Small clouds near the edge.
Edge clouds

• Mobile edge computing
• Fog computing
• Cloudlets
Edge clouds

- Mobile edge computing
- Fog computing
- Cloudlets

Rough estimate

To ensure latency requirements, more than 100 cloudlets should be needed in Europe alone!
Causality on the edge

- Datacenters + cloudlets: high number of nodes
- Partial replication
Causality on the edge

- Datacenters + cloudlets: high number of nodes
- Partial replication

**Traditional techniques to enforce causality, such as vector clocks, will not scale**
Causality on the edge

• Datacenters + cloudlets: high number of nodes
• Partial replication

Naive techniques that use small metadata may generate false dependencies
Our approach

• To leverage our previous work on Saturn
• Extend Saturn to operate on the edge
Saturn

God in ancient Roman religion, that become the god of time
Distributed metadata service

pluggable to existing geo-distributed data services

handles the dissemination of metadata among data centers

Ensures that

clients always observe a causally consistent state

with a negligible performance overhead when compared to an eventually consistency system
Metadata

more metadata    less metadata
Metadata

Matrix/vector clocks

more metadata         less metadata
Metadata

Matrix/vector clocks

One vector per item.
One entry in each vector per DC.

more metadata

less metadata
Metadata

Matrix/vector clocks

more metadata  less metadata

precise

expensive
Metadata

Matrix/vector clocks   Lamport’s clocks

more metadata   less metadata

precise

expensive
Metadata

Matrix/vector clocks

Lamport’s clocks

more metadata

precise

less metadata

expensive

One scalar.
Metadata

Matrix/vector clocks

Lamport’s clocks

more metadata

less metadata

precise

false positives

expensive

cheap
Problems of the previous state-of-the-art
Throughput vs. data staleness tradeoff

**GentleRain** [SoCC’ 14]: Optimizes throughput
Compresses metadata into a scalar

**Cure** [ICDCS’ 16]: Optimizes data freshness
Relies on a vector clock with an entry per data center
Problems of the previous state-of-the-art
Throughput vs. data staleness tradeoff

**GentleRain** [SoCC’ 14]: Optimizes throughput
Compresses metadata into a scalar

**Cure** [ICDCS’ 16]: Optimizes data freshness
Relies on a vector clock with an entry per data center

Metadata size affects throughput
Problems of the previous state-of-the-art
Throughput vs. data staleness tradeoff

**GentleRain** [SoCC’ 14]: Optimizes throughput
Compresses metadata into a scalar

**Cure** [ICDCS’ 16]: Optimizes data freshness
Relies on a vector clock with an entry per data center

Metadata size affects throughput
key features
key features

Requires a **constant and small** amount of metadata regardless of the system’s scale (servers, partitions, and locations)
Requires a **constant and small** amount of metadata regardless of the system’s scale (servers) to avoid impairing **throughput**
key features

Requires a **constant and small** amount of metadata regardless of the system’s scale (servers, partitions, locations) to avoid impairing **throughput**.

**Mitigates** the impact of **false dependencies** by relying on a tree-based dissemination.
key features

Requires a **constant and small** amount of metadata regardless of the system’s scale (servers, partitions, locations) to avoid impairing **throughput**.

Mitigates the impact of **false dependencies** by relying on a tree-based dissemination to enhance **data freshness**.
key features

Requires a **constant and small** amount of metadata regardless of the system’s scale (servers, partitions, locations) to avoid impairing throughput.

**Mitigates** the impact of **false dependencies** by relying on a tree-based dissemination.

**Implements genuine partial replication**

Data centers only manage data and metadata of the items replicated locally to enhance data freshness.
Requires a **constant and small** amount of metadata regardless of the system’s scale (servers, partitions, locations) to avoid impairing **throughput**.

**Mitigates** the impact of **false dependencies** by relying on a tree-based dissemination to enhance **data freshness**.

**Implements** **genuine partial replication** to take full advantage of **partial replication**. Data centers only manage data and metadata of the items replicated locally.
Decoupling data and metadata
Decoupling data and metadata
Decoupling data and metadata

metadata transfer

data transfer
Decoupling data and metadata

Data centers only make remote updates visible when they have received both the metadata and its corresponding data.
Example: write request
Example: write request

- data
- labels

1 2 3 ... N

data centers
Example: write request

- **data**
- **labels**

Client 1 -> 1 -> 2 -> 3 -> ... -> N -> data centers
Example: write request
Example: write request

Client 1 → put(a_1) → 1 → 2 → 3 → ... → N

data centers
Example: write request

- Client 1

- put(a₁)

- a₁

- Data centers

- Data

- Labels
Example: write request

Client 1 \( \rightarrow \) put\((a_1)\) \( \rightarrow \) 1 \( \rightarrow \) 2 \( \rightarrow \) 3 \( \rightarrow \) ... \( \rightarrow \) N

- **data**
- **labels**

Data centers
Example: write request
Example: write request

Data centers

Client 1 → put(a₁) → 1 → 2 → 3 → ... → N
Example: write request

Client 1 → put(a₁) → 1 → 2 → 3 → ... → N → data centers
Example: write request
Example: write request

Client 1 → put(a₁) → 1 → 2 → 3 → ... → N → data centers

- data
- labels
Example: write request

Client 1 → put(a1) → 1 → 2 → 3 → ... → N → data centers
Metadata dissemination graph

Saturn
Metadata dissemination graph

Saturn

\[ S_5 \rightarrow S_2 \rightarrow S_3 \rightarrow S_6 \rightarrow S_1 \rightarrow S_4 \]
The goal is to build the tree such that metadata-paths latencies (through the tree) match data-paths.

**Weighted Minimal Mismatch**

\[ \text{mismatch}_{i,j} = |\Delta^M(i, j) - \Delta(i, j)| \]

\[ \min \sum_{\forall i,j \in V} c_{i,j} \cdot \text{mismatch}_{i,j} \]
Optimal dissemination graph

The goal is to build the tree such that metadata-paths latencies (through the set of potential data-centers) to each metadata-path.

**Weighted Minimal Mismatch**

\[
mismatch_{i,j} = |\Delta^M(i, j) - \Delta(i, j)|
\]

\[
\min \sum_{\forall i,j \in V} c_{i,j} \cdot mismatch_{i,j}
\]
Optimal dissemination graph

The goal is to build the tree such that metadata-paths latencies (through the tree) match data-paths

**Weighted Minimal Mismatch**

\[
mismatch_{i,j} = |\Delta^M(i,j)| - \sum_{i,j \in V} c_{i,j} \cdot mismatch_{i,j}
\]

minimize mismatch of busiest paths
Finding the optimal tree is modelled as a constraint optimization problem.

Input

- Data-paths average latencies
- Candidate locations for serializers (an latencies among them)
- Access-patterns: to minimize the impact of mismatches
Reading

Reading/writing from the “local” datacenter is non-blocking: dependencies do not need to be checked at every operation
Reading

Reading/writing from the “local” datacenter is non-blocking: dependencies do not need to be checked at every operation.

Due to partial replication not all data is replicated locally: client needs to “migrate” to perform remote reads.
Reading

Reading/writing from the “local” datacenter is non-blocking: dependencies do not need to be checked at every operation.

Partial replication: not all data is replicated locally: client needs to “migrate” to perform remote reads.

When migrating the client may need to block: waiting for remote datacenter to be “in sync” with its causal past.
Example: migration
Example: migration

- data
- labels

1
2
3
...
N
data centers
Example: migration

- **data**
- **labels**

Client 1

1 2 3 ...

N
data centers
Example: migration

Client 1 → migrate(3) → 1 → 2 → 3 → ... → N → data centers

- data
- labels
Example: migration

Client 1 \( \rightarrow \) \text{migrate}(3) \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow N \rightarrow \text{data centers}

- Yellow circle: data
- Blue circle: labels
Example: migration

- migrate(3)

1  2  3  ...  N

data centers

Client 1

data
labels
Example: migration

- Yellow circle: data
- Blue circle: labels

Client 1 \rightarrow migrate(3) \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow N

data centers
Example: migration

Client 1

data
centers

1  2  3  …  N
Example: migration

- data
- labels

Client 1

1  2  3  ...  N

data centers
Example: migration

- data
- labels

Client 1

1 2 3 ...

data centers
Example: migration

[Diagram showing data migration between multiple data centers labeled 1 to N, with arrows indicating data flow.]
Example: migration

Client 1

1

2

3

... N

data

labels

data centers
Saturn on the edge

Challenges

- Many nodes:
  - Optimal tree may be expensive to build

- Cloudlets are smaller than datacenters:
  - Migration will be more frequent
The Saturn Rings
Let’s assume that each cloudlet stores a subset of the data maintained by a single datacenter
Let’s assume that each cloudlet stores a subset of the data maintained by a single datacenter. That datacenter is named the cloudlet’s ancestor.
The Saturn metadata tree is extended with a star of cloudlets connected to each datacenter.
This topology allows us to implement fast migration strategies.
Clients connect to the nearest cloudlet and obtain labels from the cloudlet when reading/writing data:

If a request cannot be served from the cloudlet they perform a fast migration to the datacenter (ascending fast migration).

Clients can later do a fast migration back to their local cloudlet to continue to be served locally (descending fast migration).
Fast Migration

Ascending fast migration:

Descending fast migration:
Fast Migration

**Ascending fast migration:**

Client simply presents its label (obtained from the cloudlet) to the datacenter and blocks until the datacenter is synced with the cloudlet.

**Descending fast migration:**
Fast Migration

**Ascending fast migration:**

**Descending fast migration:**

Need a little help from the Saturn brokers…
The operation of a Saturn broker is extended as follows:

When a broker ships a label to a datacenter it immediately schedules that label for transmission to the relevant cloudlet.

The broker keeps a vector with the **Last Dispatched Label (LDL)** schedule to be sent to each cloudlet $c$:

$$LDL[c]$$
**Fast Migration**

**Ascending fast migration:**

**Descending fast migration:**

Client obtains the last dispatched label (LDL) from the datacenter to its own cloudlet, presents the LDL to the cloudlet and waits for the cloudlet to be synced with the datacenter.
Migrations among siblings cloudlets:

Migration to remote datacenters/cloudlets:
Migrations among siblings cloudlets:

Requires a read operation on the ancestor datacenter.

Migration to remote datacenters/ cloudlets:
Fast Migration

Migrations among siblings cloudlets:

Migration to remote datacenters/cloudlets:

Uses the default Saturn mechanism
Provides efficient metadata management to support causality on the edged (cloudlets).

In worst case, only two labels need to be maintained by clients: a data label (used for reads/writes) and a LDL label used for fast descending migrations.