An update on Cheops: new implementation report

Geo Johns Antony, Marie Delavergne, Matthieu Rakotojaona Rainimangavelo, Baptiste Jonglez

An update on Cheops: new implementation report

Geo Johns Antony, Marie Delavergne, Matthieu Rakotojaona Rainimangavelo, Baptiste Jonglez

TECHNICAL REPORT

N° 0524

October 2024

Project-Team STACK

ISSN 0249-0803 ISRN INRIA/RT--0524--FR+ENG

An update on Cheops: new implementation report

Geo Johns Antony*, Marie Delavergne*, Matthieu Rakotojaona Rainimangavelo*, Baptiste Jonglez*†

Project-Team STACK

Technical Report n° 0524 — October 2024 — 36 pages

Abstract: This document describes the current implementation of Cheops, a software that allows to geo-distribute a cloud application by externalizing synchronization and replication of its resources. The report first summarizes the main research results around Cheops as well previous implementations of Cheops. Then, the report is focused on describing the current implementation of Cheops based on CouchDB.

Key-words: Edge Computing, geo-distribution, infrastructure software

* Nantes Université, École Centrale Nantes, IMT Atlantique, CNRS, INRIA, LS2N, UMR 6004, F-44000 Nantes, France
† Editor

RESEARCH CENTRE
Centre Inria de l'Université de Rennes
Campus universitaire de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex

Nouvelle version de Cheops : rapport d'implémentation

Résumé : Ce document décrit l'implémentation actuelle de Cheops, un logiciel permettant de géo-distribuer une application cloud en externalisant la synchronisation et la réplication des ressources de cette application. Après un tour d'horizon des résultats de recherche principaux de Cheops, puis un historique des implémentations précédentes de Cheops, nous décrivons l'implémentation actuelle de Cheops basée sur CouchDB.

Mots-clés : gestion d'infrastructures

Contents

1 Introduction - Context

Lately, there is an indubitable and growing shift from Cloud Computing to the Edge [14].

However, the assumptions that are generally taken to develop Cloud applications are not valid anymore in the Edge context. For instance, the intermittent network connections should be considered as the norm rather than the exception. If you consider the Google Doc service, users in the same vicinity cannot work on the same document if they cannot reach the data center, even though they are close to each other. Moreover, applications running in data centers

automatically assume that the connection between the different services distributed in the same data center will be stable throughout the execution of requests.

To reckon with the Edge constraints, and thus be able to satisfy requests locally, the most straightforward approach is to deploy an entire, independent instance of the application on every Edge sites. This way, if one site is separated from the rest of the network, it can still serve local requests[1]. The next step is to offer collaboration means between these instances to benefit from the entire infrastructure when needed.

Git is an application that fulfills such requirements (even though it has not been designed specifically for this paradigm): Git operations can be performed locally and pushed to other instances when required.

We proposed the premises of a generalization of these concepts by presenting how an application can be geo-distributed without intrusive changes in its business logic thanks to a service mesh approach [6]. A service mesh is a layer over microservices that intercepts requests in order to decouple functionalities such as monitoring or auto-scaling [12]. Concretely, we proposed to leverage the modularity and REST APIs of cloud applications to allow collaborations in an agnostic manner between multiple instances of the same system. In this paper, we extend our proposal to deliver a complete framework that allows multiple instances of a Cloud microservice based application to behave like a single one. Thanks to our framework, called *Cheops*, DevOps can *share*, *replicate*, and *shard* resources between the different instances in agnostic manner. A service managed by Cheops can be seen as a *Single Service Image*. Single System Images [13] is a relevant analogy as the goal of SSI clusters was based on the idea that they may be simpler to use and administer. With Cheops, the challenge related to the collaboration between multiple instances of a system (the geo-distribution aspects) is reified at the level of the DevOps and externally from the business logic of the system itself.

The contributions of this report are as follows:

- A nonintrusive approach relying on service mesh concepts to achieve three kind of collaborations between services: *sharing*, *replication*, and *cross*.
- A model of the different kind of relationships between resources that may exist in a microservice based applications.
- A detailed description of the current Cheops prototype.
- A demonstration of the feasibility of the proposed framework and collaboration strategies in the Kubernetes ecosystem.

2 Our approach

In this section, we are going to present the logic of the approach, not the way that it has been implemented, which will be in the next section.

First, we are going to present Scope-Lang, the Domain-Specific Language (DSL) we created. It is dedicated to use the collaborations between different instances of the same application across the infrastructure in an external manner. We will also give an overview of the way the solution should transfer the requests.

Second, we present how we managed to separate the synchronization logic from the business logic by differentiating the types of possible operations.

[1] We underline we do not consider disconnections between users and their Edge location. Edge elements are supposed to be as close as possible to prevent this situation.

$$
\begin{array}{lcll}
App_i, App_j & ::= & \text{application instance} & \\
s, t & ::= & \text{service} & \\
s_i, t_j & ::= & \text{service instance} & \\
Loc & ::= & App_i & \text{single location} \\
 & | & Loc\&Loc & \text{multiple locations} \\
 & | & Loc\%Loc & \text{cross locations} \\
\sigma & ::= & s : Loc, \sigma & \text{scope} \\
 & | & s : Loc &
\end{array}
$$

$$
\begin{aligned}
\mathcal{R}[\![s : App_i]\!] &= s_i \\
\mathcal{R}[\![s : Loc\&Loc']\!] &= \mathcal{R}[\![s : Loc]\!] \text{ and } \mathcal{R}[\![s : Loc']\!] \\
\mathcal{R}[\![s : Loc\%Loc']\!] &= \mathcal{R}[\![s : Loc]\!] \text{ spread to } \mathcal{R}[\![s : Loc']\!]
\end{aligned}
$$

Figure 1: Scope-lang expressions σ and the function that resolves service instance from elements of the scope $\mathcal{R}$.

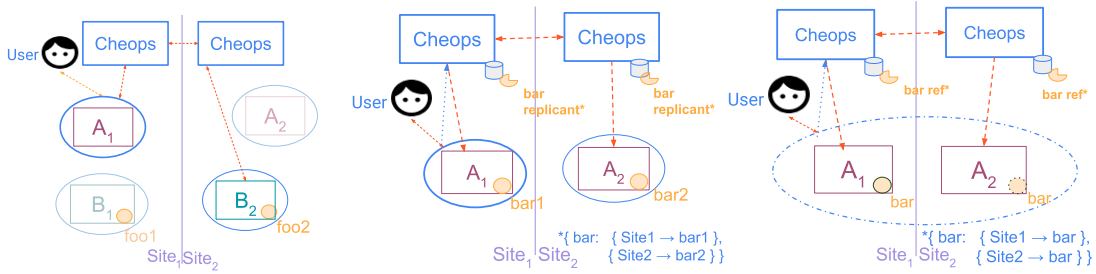
2.1 Towards generic and noninvasive collaborations

2.1.1 Scope-lang

Scope-lang is a language introduced in [6], that extends the usual requests made from users to their application in order to reify the locality aspects. It can be used in a call to the application natively or be wrapped in a requests directed to the service mesh/chosen solution. A scope-lang expression, which we call *scope*, contains information on the location where a specific request, or part of the request, will be executed. As an example, a scope defined as "$s : App_1, t : App_2$" specifies to use service s from App_1 (the application *App* on Site 1), and the service t from App_2 (on Site 2). Users defines the scope of the request to specify the exact collaboration between instances required for the execution of their request. The scope is interpreted during the execution of the request workflow to decide the execution location accordingly. A more formal definition of the language is available in Figure 1, which has been extended to allow cross collaborations.

2.1.2 Collaboration implementations for elementary resources

Figure 2 depicts the three collaborations considered for *Cheops*, the solution we are going to implement as a service mesh comprised of a proxy to intercept requests and the service to geo-distribute these resquests.



(a) Sharing (b) Replication (c) Cross

Figure 2: The different Cheops collaborations

2.1.3 Sharing

Sharing is the collaboration which allows a service instance to use a resource from a service which is not the one assigned to its application instance.

The typical example is getting a resource from a service B on another site for a service A as presented by the red arrows in Figure 2a. :
`application create a --sub-resource foo2 --scope {A:` $Site_1$`, B:` $Site_2$`}`

1. A user requests to create a resource on service A from $Site_1$ (Service A_1), using a sub-resource foo_2 from service B on $Site_2$ (Service B_2).
2. The request is intercepted and transferred to Cheops.
3. Cheops extracts the scope from the request and interprets it.
4. Cheops transfers the request to service A, until A needs the sub-resource.
5. The outgoing request is intercepted, and at this point, is transferred to $Site_2$.
6. Cheops on $Site_2$ uses its catalog to find Service B endpoint and transfer the request to this service to get $foo2$.
7. The service response (containing the resource itself) is finally transferred back to Service A through Cheops.

2.1.4 Replication

Replication is the ability for users to create and have available resources on different Edge sites to deal with latency and split networks. Replication main action is duplication: transfer the request to every involved sites and let the application execute the request locally. The operation does not simply consists in forwarding the request to the different instances, though. Cheops keeps track of the different replicas in order to ensure that future CRUD operations achieved on any replica will be applied on all copies, maintaining eventually the consistency over time. To do that, Cheops relies on a data scheme, called the replicant, that links a meta-ID to the different replica IDs and their locations.

Figure 2b sums up the workflow to create a replicated resource on two sites:
`application create a --name bar --scope {A:` $Site_1$ `&` $Site_2$`}`.

1. A user sends a request on $Site_1$ to create two replicas of the *bar* resource.
2. The request is intercepted and transferred to Cheops.
3. Cheops extracts the scope and interprets it.
4. Cheops creates the replicant, and passes the request to create it to Cheops on the other involved site ($Site_2$), as well as the request of creation of *bar* on both sites, which is simply the request without the scope.
5. Both Cheops execute the request of creation; the response is intercepted to fill the local IDs on the replicants and the response is transferred to the user, replacing the local ID by the meta-ID of the replicant.

To provide eventual consistency, Cheops follows the Raft protocol, with one replicant acting as the leader.

2.1.5 Cross

Cross is the last collaboration we identified.

The idea is to create a resource over multiple sites. The main difference with respect to the aforementioned replication concept is related to the aggregation/divisibility property. In the replication, each copy is independent, even if they all converge eventually based on the CRUD operation. A cross resource can be seen as an aggregation of all resources that constitutes the cross-resource overall. Some resources which cannot be divided by an application API will require an additional layer in the business logic to satisfy the divisibility property.

Similarly to the replicant data scheme, Cheops keeps tracks of the different resources in order to perform CRUD operations in the expected manner. A `CREATE` operation for instance can distribute the resource over different sites (if this resource is divisible), while a `READ` will be performed on each "sub-resource" composing the cross-resource in order to return the aggregated result.

How Cheops deals with split-brain issue for cross resource is left as future work. However, it is worth noting that the unreachability of one site that hosts a part of the cross resource faces multiple challenges. An illustration of Cross is depicted in Figure 2c:
`application create a --name bar --scope {A:` $Site_1$`%` $Site_2$`}`.

1. A user sends a request to create a resource specifying the involved sites.
2. The request is intercepted and transferred to Cheops.
3. Cheops extracts the scope and interprets it.
4. Cheops creates the resource on the first site ($Site_1$) and passes the request to other involved sites.
5. Cheops on $Site_2$ identifies the extended resource and creates an identifier within Cheops to forward to the deployed resource site

The complete explanation will be available in Geo Johns Antony thesis manuscript[2].

2.2 An External Approach Toward Consistency

Most existing consistency models, such as Paxos and RAFT, require global consensus to synchronize state across geo-distributed instances. While these protocols ensure strong consistency, they can introduce significant latency, especially in network partition scenarios where global agreement is delayed or temporarily unavailable. As geo-distribution becomes more common in modern cloud-native and edge applications, the trade-off between availability and consistency becomes increasingly important. The approach proposes an external consistency model that manages consistency without embedding the logic directly into the application. This approach works alongside geo-distributed instances, preserving strong eventual consistency (SEC) while prioritizing the local-first principle. In this model, operations on resources can be processed locally, even if the system is temporarily partitioned, and the state converges once the network is restored. By externalizing consistency, the system remains flexible, scalable, and compatible with a wide range of applications.

This approach avoids forcing applications to adopt specific consistency protocols internally, making the solution generic and non-intrusive to any application architecture. It allows developers to focus on the core business logic without worrying about how to synchronize states between geo-distributed instances. Moreover, this solution supports heterogeneous systems, meaning different consistency strategies can be applied to different types of resources within the same application.

2.2.1 Extending Stateful Operations

A key aspect of this consistency model is handling stateful operations in a geo-distributed setting. A stateful operation affects the system's state and may cause conflicts if executed concurrently in different locations. To manage these operations, the chapter presents a method of categorizing operations based on their behavior and interaction with the system's state.

The external consistency model classifies operations into two broad categories:

[2] https://theses.fr/s280363

Replace Operations: These operations completely replace the current state with a new value. Since replace operations are idempotent (i.e., applying them multiple times yields the same result), they are easier to manage in a geo-distributed context. Examples include setting a specific configuration value or resetting a counter to zero.

Iterative Operations: These operations modify the current state based on its previous value, and they are non-idempotent. For instance, incrementing a counter or appending to a log file are iterative operations that depend on the current state. These are more complex in distributed systems because concurrent modifications can lead to inconsistencies if not handled correctly.

2.2.2 Classifying Operations for Conflict Resolution

To effectively manage stateful operations, Cheops categorizes concurrent operations into three distinct classes, each with its own strategy for conflict resolution. These classes allow Cheops to handle concurrent operations in a non-intrusive manner, external to the application logic, ensuring continued operation even during network partitions. Operations are applied locally at each site, and the approach is flexible enough to be applied to any application without requiring changes to the core code.

Class 1: Iterative & Commutative Operations In Class 1, operations are iterative (they change the state incrementally) and commutative (the order of the operations does not affect the final outcome). This class typically includes operations like increments or decrements in counters (e.g., a Positive-Negative Counter or P-N Counter), where operations can be applied in any order, and replicas will still converge to the same final value. For example, two concurrent operations, $increment_counter(10)$ and $decrement_counter(5)$, executed at different sites will result in the same final value, regardless of the order in which they are applied. This is because the operations are commutative and iterative. Cheops ensures these operations are applied exactly once per replica using a Reliable Causal Broadcast (RCB) protocol. The RCB protocol guarantees that all operations are delivered to every replica, ensuring that no operation is applied multiple times. As a result, operations in Class 1 converge naturally across all replicas due to their commutative nature.

Class 2: Replace & Non-Commutative Operations Class 2 includes at least one replace operation, which overwrites the current state, and any combination of replace and non-commutative operations. In contrast to Class 1, the order of execution in Class 2 is critical for achieving convergence. Non-commutative operations mean that the final result depends on the order in which operations are applied across replicas. For example, consider two concurrent operations on a replicated string resource: $set_string("foo")$ and $append_string("bar")$. These operations are non-commutative because the order in which they are applied determines the final value (either foo or $foobar$). To resolve conflicts in Class 2, Cheops uses a Last-Writer-Wins (LWW) strategy, where the most recent operation, based on a timestamp, is applied at all replicas. This ensures that even non-commutative operations are executed in the same order across all sites, leading to eventual consistency. In some cases, however, LWW may result in losing certain updates (e.g., discarding an append operation). Cheops provides additional flexibility in handling such scenarios by allowing developers to define specific rules or strategies for conflict resolution that better suit their application's needs.

Class	Class 1	Class 2	Class 3
Resulting combination of operations	Iterative & commutative	Replace & non-commutative	Iterative & non-commutative
Examples	Two increments in a PN counter	Append & set in a string	Two append operations in a string
Resolution strategy (RCB by default)	Apply once everywhere	Precedence order	Explicit resolution with resource logic

Table 1: Each class represents a resulting combination of concurrent operations and their corresponding strategies for conflict resolution.

Class 3: Iterative & Non-Commutative Operations Class 3 involves iterative operations that are non-commutative, meaning that the operations depend both on the current state and the order in which they are applied. This class includes more complex scenarios, such as concurrent appends and left shifts in a string resource or operations like addition and multiplication on a number. For example, consider a string resource with the initial value "foo", and two concurrent operations: append("bar")and left_shift(2). At one site, the append("bar") operation would result in "foobar", while at another site, left_shift(2) would result in "oof". Simply using LWW to resolve these conflicts would not be sufficient, as one of the operations would be lost, leading to inconsistent states across replicas.

To solve this, Cheops introduces *Consistency Logic*, an external mechanism that allows users to define custom resolution strategies for Class 3 operations. Cheops integrates this approach with external algorithms, like Peritext, which is designed for handling concurrent operations in collaborative text editing. By externalizing *Peritext*, Cheops uses metadata and unique operation IDs to ensure that conflicting operations are consistently ordered and applied at all replicas, ensuring convergence.

Operation Metadata: Each operation is tagged with metadata that includes a unique identifier (site ID and logical clock). This metadata helps determine the correct order of operations across replicas.

Commutativity and Associativity: Operations are designed to be commutative and associative, meaning their order can be rearranged without affecting the final result, ensuring convergence.

Handling Concurrent Operations: When two operations (e.g., appends or shifts) are concurrent, Cheops applies them in a consistent order using the metadata. This guarantees that all replicas end up with the same final result.

Through consistency logic, Cheops externalizes this conflict resolution process, making it possible to adapt algorithms like Peritext to the distributed system. Furthermore, any other algorithm designed for conflict resolution, whether it involves commutative data types like *CRDTs* or custom strategies, can be plugged into Cheops via consistency logic. This ensures that Class 3 operations, which are often the most complex and prone to divergence, can achieve strong eventual consistency while remaining flexible for different application needs.

These classes and their resolution strategy are summarized in 1.

By classifying operations and applying resource-specific conflict resolution strategies, Cheops ensures *strong eventual consistency* while ensuring a local-first operation at each site and being generic and non-intrusive to any application.

2.2.3 Mapping Operations to Classes

Once the operations are classified, the chapter explains how Cheops (the middleware managing geo-distributed consistency) can map each operation to the appropriate class and apply the correct conflict resolution strategy. Cheops addresses the challenge of handling concurrent operations on geo-distributed resources by using a *consistency matrix*. This matrix is constructed for each resource type, listing all possible operations along its rows and columns. The matrix maps every combination of concurrent operations to a specific class of resolution strategy, which defines how conflicts should be handled.

The approach is designed to be generic, allowing Cheops to apply the matrix to any resource type by using user-provided inputs to map each combination of operations to a corresponding class. For instance, in the case of a P-N Counter, the operations might include increment, decrement, and set. Cheops uses the matrix to determine how to resolve conflicts when these operations occur concurrently. For example, if an increment and a decrement are concurrent, *Class 1* semantics are applied, ensuring both operations are executed once at each replica.

For more complex resources, like strings, additional operations like *LeftShift*, *RightShift*, *Append*, and *Delete* require more nuanced conflict resolution strategies. In these cases, the matrix assigns classes that reflect whether operations are commutative, iterative, or replace-based. Similarly, for *Kubernetes Pods*, where operations involve configurations, a deterministic approach like *Last-Writer-Wins (LWW)* is used. In these cases, all concurrent operations fall under *Class 2* semantics, simplifying conflict resolution. This matrix-based solution provides Cheops with the flexibility to handle various resource types and operation combinations, ensuring that concurrent conflicts are resolved effectively and geo-distributed replicas converge correctly.

Handling Exceptions When Eventual Convergence Fails: The Case of Concurrent Operations In distributed systems, ensuring consistency across geo-distributed instances becomes especially challenging when operations are concurrent and eventual convergence fails. Cheops addresses this issue with a novel *hooks-based approach* that extends beyond the traditional consistency matrix to detect and resolve conflicts specific to the resource's internal constraints. This challenge is illustrated using the example of a *Positive-only Counter (P-Counter)*, where the value must not fall below zero.

P-Counter Conflict Scenario: Consider a replicated P-Counter with an initial value of 100. Two concurrent operations occur: Replica 1 issues a *decrement*(50) operation, reducing the counter to 50. Replica 2 simultaneously issues a *decrement*(51) operation, reducing the counter to 49. Locally, both operations succeed because the decrements do not violate the counter's constraints at their respective sites. However, when the operations propagate to the remote replicas, a conflict arises: applying both decrements results in a value below zero, which violates the P-Counter's core rule. This is a conflict that the standard consistency matrix cannot resolve, as it is due to an internal resource constraint rather than a simple concurrency issue.

Cheops Hooks Approach To address such cases, Cheops employs a *hooks mechanism*. This mechanism allows users to define custom error detection and resolution strategies specific to resource constraints, extending Cheops' non-intrusive nature. When the P-Counter's value drops below zero, the system captures an error through its *partial error mechanism*.

Error Detection: When the decrements propagate and cause the P-Counter to fall below zero, Cheops collects these errors using its partial error mechanism. The error is flagged as a violation of the resource's structure, i.e., the P-Counter rule that it cannot have a negative value.

Conflict Resolution: Cheops then triggers a user-defined resolution strategy via the hooks logic. In this case, *operation decomposition* is used. Instead of applying the full decrements (50 and 51), the operations are broken down into smaller increments (e.g., $50 \times decrement(1)$), which are applied sequentially across both replicas.

Ensuring Convergence: As each operation is applied, Cheops monitors the state of the counter. When the value is about to drop below zero, the system halts further decrements and signals the conflict. This ensures that the final value converges correctly across all replicas without violating the P-Counter's integrity.

Example Workflow

1. A P-Counter starts at 100, and concurrent operations $decrement(50)$ and $decrement(51)$ are issued from replicas 1 and 2, respectively.

2. Each operation succeeds locally but results in an error when propagated to the other site, as the combined decrements would reduce the value below zero.

3. Cheops detects this error via its hooks mechanism, which triggers the user-defined resolution logic.

4. The operations are decomposed into smaller steps (e.g., $decrement(1)$), and applied sequentially across both replicas.

5. Once the value reaches zero, Cheops stops further decrements, ensuring consistency while preventing a negative counter.

Conclusion on the mapping This hooks-based solution enables Cheops to handle exceptions and conflicts that go beyond typical concurrent operation issues. By allowing users to define resource-specific error handling and resolution strategies, Cheops maintains strong eventual consistency even in complex cases like the P-Counter, where internal resource constraints must be respected. This approach ensures that geo-distributed instances converge correctly, even when network partitions or concurrent operations introduce conflicts.

2.2.4 Validation and Use Cases

The proposed external consistency model is validated through several real-world use cases, primarily focusing on geo-distributed environments involving Kubernetes and OpenStack: Kubernetes Pods and Deployments: Cheops is used to manage the state of Kubernetes Pods across multiple clusters, ensuring that changes such as scaling operations or image updates are propagated consistently. The system leverages the classification matrix to apply appropriate conflict resolution strategies based on the type of operation. For Kubernetes Deployments, the consistency model ensures that configuration updates, scaling factors, and deployment states converge to a consistent final state across all instances.

2.2.5 Conclusion on consistency

We prresented here a comprehensive framework for managing consistency across geo-distributed applications by externalizing consistency management. This approach significantly reduces the complexity for developers, allowing them to focus on core application logic without worrying about the intricacies of state synchronization across multiple locations. By categorizing operations into classes and applying the appropriate conflict resolution strategy, Cheops ensures that geo-distributed instances converge to a consistent state without sacrificing availability or performance. Through detailed validation with Kubernetes and OpenStack, the chapter demonstrates the flexibility and scalability of this solution, which is poised to handle the increasing complexity of modern, geo-distributed applications.

To know more about this framework, as well as how to handle dependencies between resources externally, refer to the contributions in Geo Johns Antony's manuscript[3].

3 Our implementation

Cheops' goal is to sit between the user and the different instances of the application so that it can automatically distribute the desired intention to all application instances.

Cheops went through different entire overhauls following potential improvements based on theoretical enhancements.

The first version was a test to use the Consul[4] service mesh and Envoy[5] as a reverse proxy. As sharing was the first collaboration implemented for OpenStack[6] in openstackoid [5,8], the goal of this Proof of Concept was to allow sharing on different plaftforms using a service mesh that was already widely used. It was made by Matthieu Juzdzewski, Arnaud Szymanek, Ronan-Alexandre Cherrueau and Marie Delavergne under the supervision of Adrien Lebre. For more information on this version, refer to the Appendix in [10].

Afterwards, the work followed on a more light and tailored to our need approach, developed by Geo Johns Antony and Marie Delavergne under the supervision of Ronan-Alexandre Cherrueau and Adrien Lebre. This version used HaProxy[7] to communicate between different instances of Cheops and ArangoDB[8] as a database to store the meta informations required for replication and cross. This Proof of Concept is more detailed in [11].

Further versions tried to look at the state of the application and replicate that to other nodes, effectively doing continuous live migrations. The issue with that thinking is that most applications do not have a way to dump their state or restore from a given state: that approach was not fruitful.

The current implementation has been developed by Matthieu Rakotojaona Rainimangavelo, Marie Delavergne and Geo Johns Antony. The core tenet is the idea of intercepting all commands given by the operator to the application, and replicate them. A command here is very specifically a shell command targeting a specific resource and can be executed. We package those commands into *operations*, store them and distribute them. The approach created heavy changes in the approach on classification and consistency.

Such *operations* can take any form the application uses to manage resources:

- For kubernetes it can be a command to create a pod such as

[3] https://theses.fr/s280363
[4] https://www.consul.io/
[5] https://www.envoyproxy.io/
[6] https://www.openstack.org/
[7] https://www.haproxy.org/
[8] https://arangodb.com/

```
$ kubectl create pod
```

or a command to apply a specific recipe:

```
$ kubectl apply recipe.yaml
```

- For redis it will be a command to set a specific value

```
$ redis-cli set variable 13
```

or one to add an element to a set

```
$ redis-cli sadd my-set 12
```

- For a filesystem, a command can be the modification of a file:

```
$ echo "content" > new-file
```

These operations are assumed to contain enough information to modify the resource as the operator wants, and as such must be used by the operator. The job of Cheops, then, is to orchestrate operations between nodes and between each other so as to converge towards a consistent state of the resource. The current implementation assumes that all resources are independent from each other: future work will focus on relationship and dependencies between resources, and how they have to be targeted for more workflows to work. It alos assumes that operations go through Cheops, and doesn't try to guess any change that may have been done directly. Obviously the operations Cheops manages will be modifying operations: operations that do not affect the state have no conflict problems.

Diagram Figure 3 shows the high-level view of how Cheops is architectured. Cheops organizes independent nodes with fluctuant networking and availability conditions, represented by the boxes. Each node has 3 elements: a CouchDB instance, the application to geo-distribute, and the Cheops process itself sitting in the middle.

3.1 CouchDB, and how we use it

CouchDB [9] is an open-source JSON Document database focused on replication. It has a proven track record of being a reliable tool to efficiently synchronize data and this is why it why chosen. This efficiency and reliabily is provided by CouchDB implementing Multi-Version Concurrency Control (MVCC): all documents are associated with a revision, and a modification of a document must be associated with a parent revision by the user.

At the top-level, CouchDB implements *databases*, which are just a collection of documents with a specific access model. Since we completely trust it, we have a single *database* where Cheops has full access. In the remaining paragraph we will assume only one database per node and when we talk about documents, we talk about documents in that all-encompassing database.

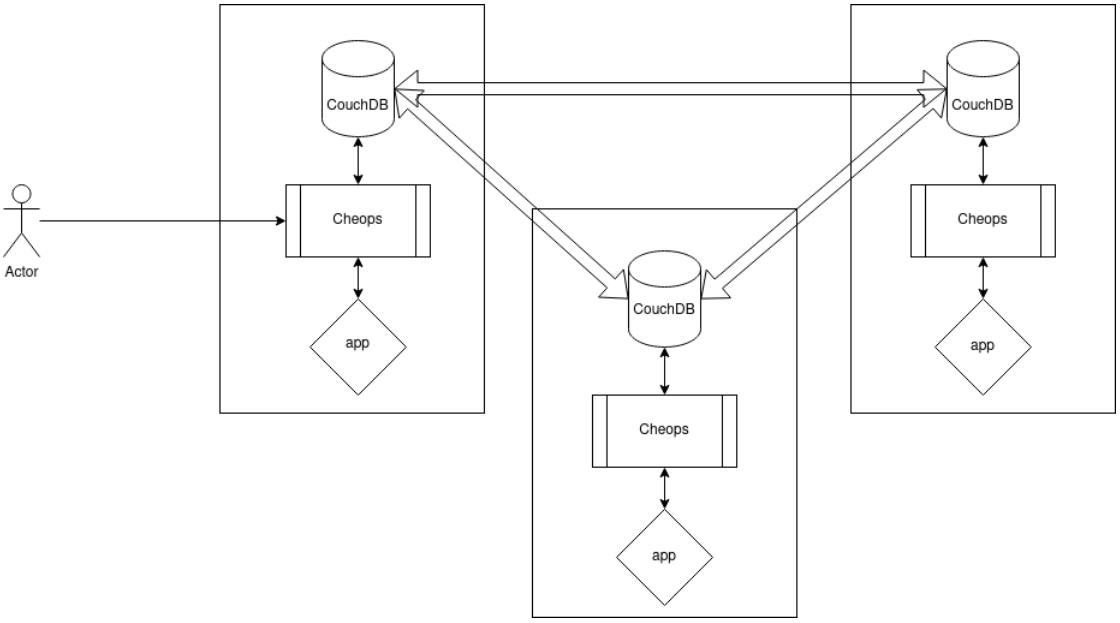


Figure 3: High-level view of Cheops nodes and how they interact

CouchDB implements a custom protocol to efficiently replicate the full tree of revisions for each document between CouchDB nodes. After replication is done, thanks to MVCC, it is possible to deterministically agree on a common revision on all nodes: in effect it is a variation of Last-Writer-Wins. It is also possible to know whether there are conflicts. Those two properties will be at the core of our conflict management because they give us (i) a common *winning* revision at all times and (ii) a list of potential conflicts. It is important to note, however, that this revisions scheme is not a full versioning system such as git: it is only a system to help the user detect potential conflicts. CouchDB makes no guarantees that earlier versions are still available, and indeed garbage-collects them automatically. Note that garbage collection never happens on *live* revisions, that is, revisions that are leaves of the graph and thus the *winning* revision with its potential conflicts.

A CouchDB replication job always has a source and a target database, and sends documents and versions that exist on the source but not on the target. Both source and target can be local or remote *databases*, but as we stated earlier we only use one database per node: the source and the target will in practice be two different nodes at all time. For a bidirectional replication, 2 jobs have to be created, one in each direction. Whatever the configuration, replication are managed by jobs so they can be watched, paused, or removed. Any number of jobs can exist. This allows versatile options for replication:

- replication can be done once until completion or continually: for our use-case we will use the continuous mode
- if the source is a local database, CouchDB has efficient mechanisms to automatically see changes and potentially replicate them. If the source is a remote database, it has to be polled. For efficiency we will only watch local databases, from each node, and send to remote databases

By default a replication job will send all documents, but it is possible to specify a specific

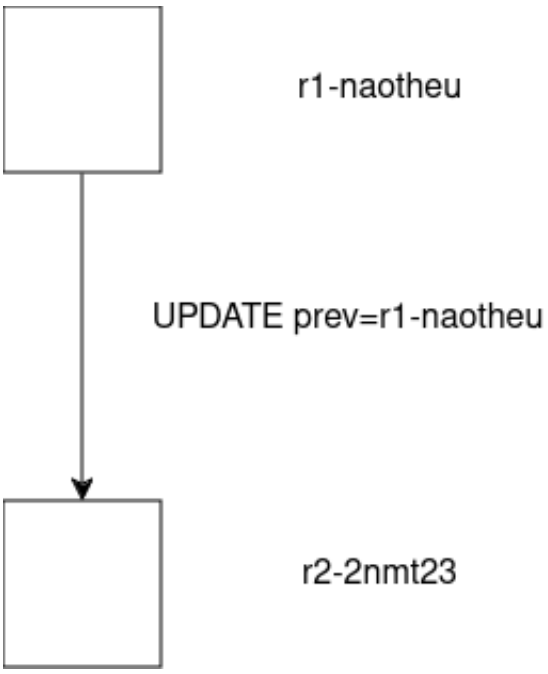


Figure 4: In CouchDB, updating a document requires setting the previous revision

query that will be tried on each revision of each document; if it matches, the revision is sent, otherwise it is not handled.

CouchDB was created during the hype era of NoSQL and as such experimented with other ways of querying data: in its case Map-Reduce, a way for the user to extract values about documents and optionally extract a meta-value out of it. Later on CouchDB also implemented the MongoDB query language, more natural to users. Finally CouchDB provides an endpoint that can be continuously used to detect changes to the set of documents: a new document is created or a document is changed (the set of its revisions change)

To ensure proper replication without losing any data, CouchDB implements an MVCC[9] strategy: every document is associated with a revision string decided by CouchDB itself. The user has no control over it. When the user wants to update a document, they know the previous version *and* its revision string and must pass it along with the new version, in effect creating a directed edge, as can be seen in Figure 4; CouchDB gives the new revision a new number. The architectural model is thus not a mapping from a key to a document but from a key to a directed acyclic graph of documents, each with its own revision string, with the root being the very first version.

When two nodes update the same document, each node will give a different revision; we will thus end with multiple child revisions at the same level, as shown in Figure 5. The entire graph is reliably synchronized by the replication protocol: each version of a document being immutable, it is easy to see a full replication will not step on its own toes.

At this point the document has two versions, but from a user point of view the key should map to a single version: CouchDB uses an internal but deterministic algorithm to determine a *winning* version, in a Last-Writer-Wins fashion. Since the algorithm is deterministic, given the

[9] https://en.wikipedia.org/wiki/Multiversion_concurrency_control

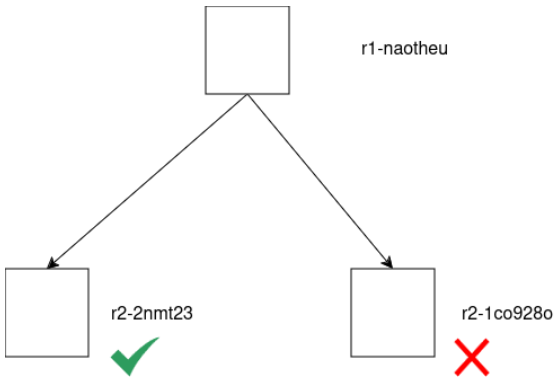


Figure 5: Updating the same document from two different places. CouchDB selects one revision as winning, the other as a conflict

same tree, all nodes will always return the same version. However since there are other versions, on top of getting a *winning* revision, CouchDB can tell the user the document has conflicts, and give them all the conflicting versions if they so desire. In the simplest case it is fine to continue as-is, but the user has the possibility to enforce a good revision by marking all the others as deleted. This deletion mark is then replicated. Resolution algorithm is entirely the business of the user and can involve picking one of the existing revision or doing a local merge and updating with a new, merged revision. In Figure 6 we see the user marked a version as deleted and updated the other one with a new version, perhaps following a custom merging strategy.

Two important details are important to note here:

- The user can only update leaf revisions: they are the only revisions alive. It is not possible to update by setting as previous an earlier revision. In the example here the first revision cannot be used as a parent *except* if the node only knows it

- It doesn't really matter what version is the parent: from the point of view of the user the lineage has no meaning. Whatever parent is used in Figure 6 the result will be the same: there is only one version

The consequence of both points is that in practice users have no access to earlier revisions. This mechanism is not a way for them to keep a history of modifications, it is *only* a way to deterministically get a winning revision on all nodes. In fact CouchDB periodically removes the content of earlier revisions, keeping only the revision strings and their parent, to save space. This mechanism, known as compaction, is configurable but it is not advised to disable it.

Note that if two independent nodes update with exactly the same content with exactly the same ancestry, then CouchDB will give the new version the same revision string. Upon synhcronization it will see that that revision is already synchronized, and not create anything more: there will be no conflict, only a single revision. This is useful because it eliminates the situation where the user thinks there is a conflict when the content is actually exactly the same: CouchDB just does the right thing.

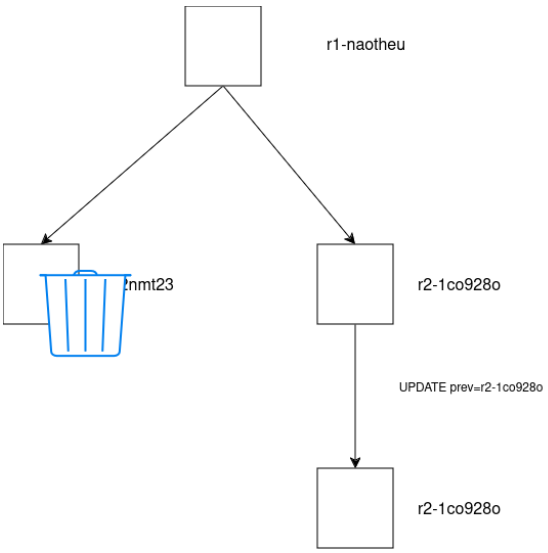


Figure 6: User marked a revision as deleted and added a new one: it is now the only version of the document, with no conflicts

3.2 On top of it: Cheops

Cheops is a custom software implemented in slightly over 2000 lines of Go code. It offers an HTTP API[10] to users who can send operations along with the id of the resource and the sites where the resource is expected to be Cheops does not store the state of the resource, which is kept at the application level: it only deals with operations.

Figure 7 shows the elements composing the Cheops stack: the API receives operations from the actor, the Replicator sends them to CouchDB for replication and manages conflicts resolution, and the Backend executes operations.

The Backend is a thin layer interacting with the application being managed. It is responsible for running shell commands and returning the reply and status from the application. There is no security and no checks involved: the command is run as-is, with the user and environment of the Cheops process. The command must exist, Cheops isn't responsible for installing it. Any shell command valid for an application is accepted:

- kubernetes is typically used through the *kubectl* command:

```
$ kubectl create pod
```

- redis is used through its *redis-cli* command:

```
$ redis-cli hset players-age jane 27
```

3.2.1 Operations (Re-)ordering

Cheops creates an implementation of Reliable Causal Broadcast [2] by creating a directed acyclic graph with operations as nodes and causality as edges.

[10] https://gitlab.inria.fr/discovery/cheops/-/blob/main/openapi.yml

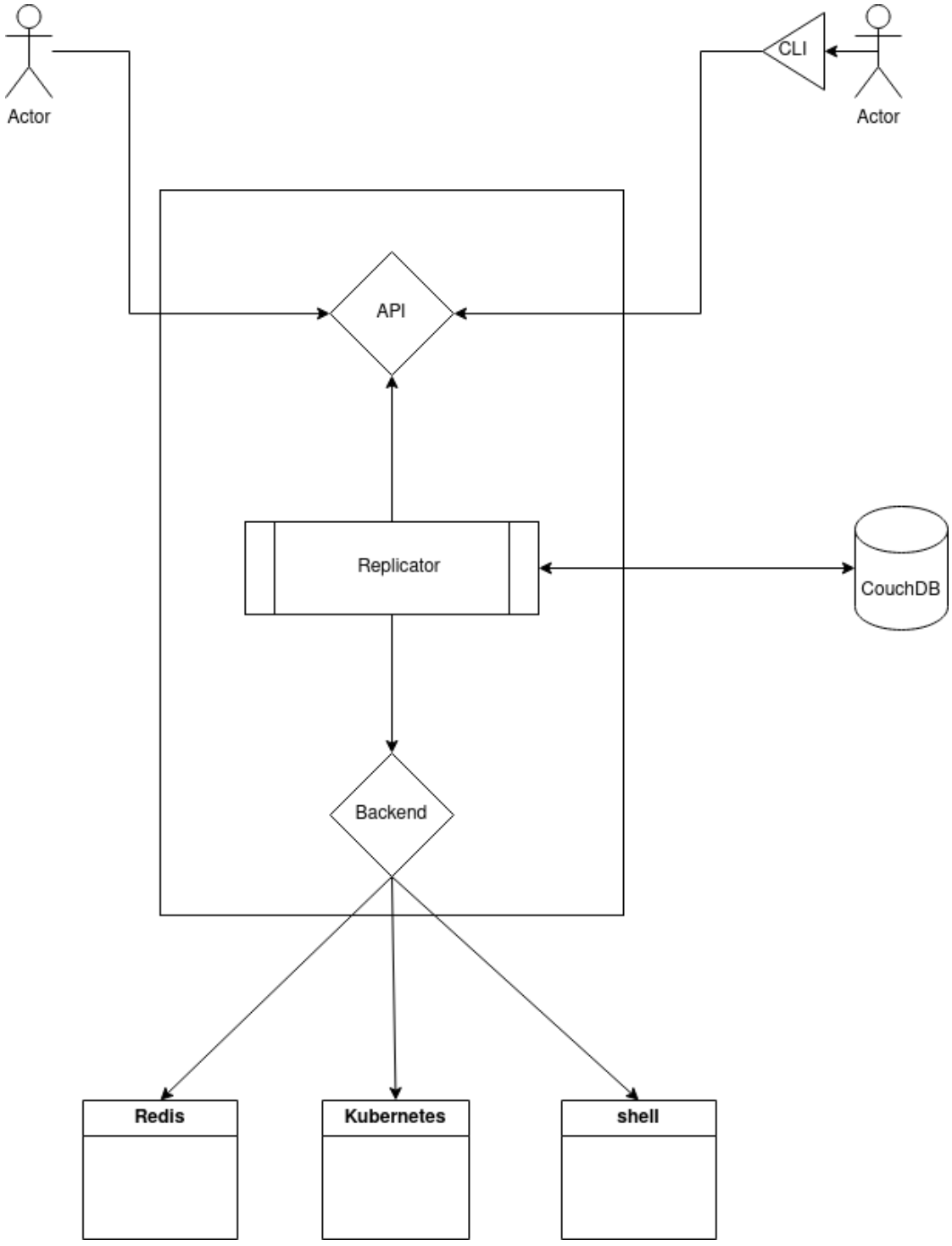


Figure 7: Zoomed-in view of Cheops components and how they interact with the exterior

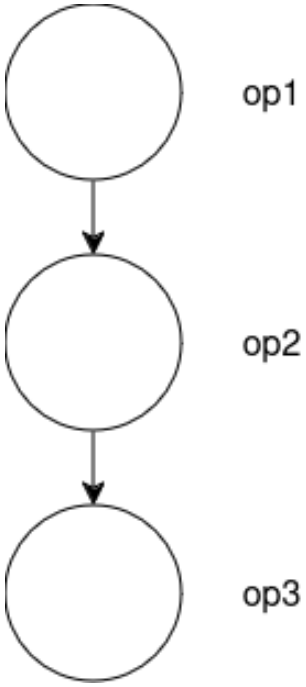


Figure 8: A list of 3 operations

Figure 8 shows how operations follow each other, in order. Operations are examined when they are received, and appended on the existing list. This is done on the site where the operation is received, without any coordination with other edges. Some metadata is added to each operation, such as the site that received it, the time it received it, and a unique identifier.

An edge between two operations is always directed from the operation happening before to the operation happening after. As such, 2 operations may not have a direct edge, but still be indirectly related through parentage: in Figure 8 *op1* is an ancestor of *op3* even though there is no direct link. There is a causal relationship here: *op3* is causally related to both *op2* and *op3* happening before, in that order.

As an optimization, since some operations are of type replace, involving them in a *Class 2* relationship, it is not necessary to remember what happened before because it will be overwritten. The other consequence of this is that any operation that follows a replace operation but is not itself of type replace gets appended; what we end up is a list of operations where **the first operation is probably of type replace, and every other operation after is not**. Adding a new operation on top of that list is then a matter of comparing only with the last operation, to know if the new one will replace everything or be appended. Moreover, since we haven't tackled the *Class 3* pairs at the moment, we know that **all operations following the first one will necessary be commutative**

This list of operations is stored as a CouchDB Document, along with the list of sites where the resource is supposed to exist. Every time that document is modified, it is then broadcasted to all other nodes, who will update their local vision of the resource by replacing the list of operations with the new one. In the most common case where a single operation was added, each node only has to run that new operation to arrive to the same resource state. Even if the user added multiple operations it will be one straightforward.

3.2.2 Conflicts

Of course, that is not always so simple, and not what we aimed to solve: sometimes multiple nodes added operations at the same time. CouchDB synchronizes all versions of the document, and sees that they are in conflict, as seen in the previous section. As we have also seen CouchDB can give us a *winning* revision and the list of all conflicts along with all their content, that is when we know there is a conflict in the list of operations to apply.

As a first step in resolving conflicts we must understand how multiple operations work with each other and how they can be handled as concurrent. To do this we build a matrix to classify pairs of operations, as described in the previous chapter. Thanks to this classification, Cheops knows what to do when multiple operations are concurrent. For example, Table 2 is a matrix for an integer managed by two redis operations: *SET*, resetting its value, and *INCRBY*, incrementing it by a given value (potentially negative). adds at the end of the file.

Operations	SET	INCRBY
SET	Class 2	Class 2
INCRBY	Class 2	Class 1

Table 2: Operations matrix for an integer

This table shows two situations:

- *SET* operations are of type replace, so any pair involving them will be of *Class 2*
- *INCRBY* are commutative with each other, so a pair of both *INCRBY* operations will be of *Class 1*

Table 3 is an *operations relationship matrix* for filesystem operations on files. > is an operation that replaces the content of a file, and >> is an operation that appends at the end of the file.

Operations	>	>>
>	Class 2	Class 2
>>	Class 2	Class 3

Table 3: Operations matrix for a file with two commands

There are, again, multiple cases:

- > operations are of type replace, so they will always be of *Class 2* when paired with any other command
- >> append content at the end of the file; how to combine them ? There is no clear solution and they need additional logic. The pair is of *Class 3*

Recall that in any list of operations, the first operation might be of type replace, and all the others will never be and will be commutative with each other. They only need to be kept in the same order, because they are iterative meaning they depend on the state before them. Thanks to that we see that a proper resolution algorithm of the list of operations will involve comparing the first operation of the list from the winning revision and the first operation of the list from the first conflict, check what Class is the pair, make a decision about which of the operation of the pair to keep in the final list (including, maybe, both) and then keep all other operations, as they are commutative with each other

- If both first operations on each side are commutative, i.e., *Class 1*, then it is ok to keep both operations in any order
- If both first operations on each side are of type replace (*Class 2*), then only one should be kept, deterministically
- If one operation is of type replace and the other is not (still *Class 2*), then we keep the one of type replace

Once a new list is constructed we then re-iterate the algorithm with every other conflict until they have all been handled. We end up with a new list of operations that is the *merge* of all conflicting versions that we then save, and mark all others as deleted. CouchDB will then re-synchronized this new list along with the deletions. Because this algorithm only considers the content that will eventually be the same on all nodes, it is deterministic, and as such if run on multiple nodes will end up with the same result, and CouchDB will do the right thing and not consider this a new conflict.

As we saw there are two scenarios to handle for *Class 2* pairs, and this needs to be encoded in a way that differentiate the two cases.

3.2.3 Operations resolution matrix

As we have seen the operator needs to specify multiple scenarios regarding operations and how they work together:

- Whether adding a new operation will actually add it at the end, or replace everything
- How to handle conflicting operations

To solve this crucial configuration problem we have established a new matrix called the **Operations Resolution Matrix** to be specified by the operator and tells Cheops how to behave. This matrix is a list of resolution entries. Each intent is composed of two operations, in order, and the result to apply. Operations are noted as *Before* and *After* to remember the order. The goal of the matrix is to know which of the operations to keep based on their order, so the result says which one to *take*, i.e., keep. Results are one of four possibilities:

take-one : Only one operation can be taken, but it can be any of them

take-both : Both operations can be taken in any order

take-both-keep-order : Both operations can be taken only if they are in this order

take-both-reverse-order : Both operations can be taken only if they are in the reverse order

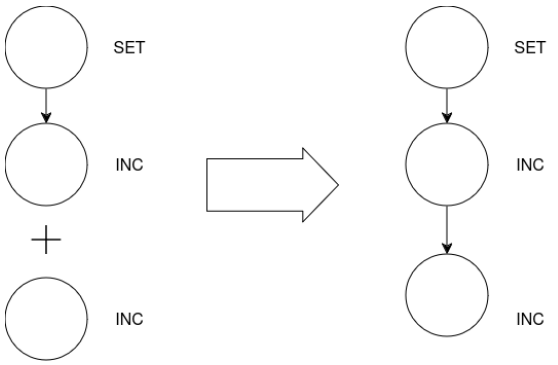


Figure 9: Adding an INC operation on top of an INC operation when the pair (INC, INC) has resolution *take-both* results in the new operation being appended

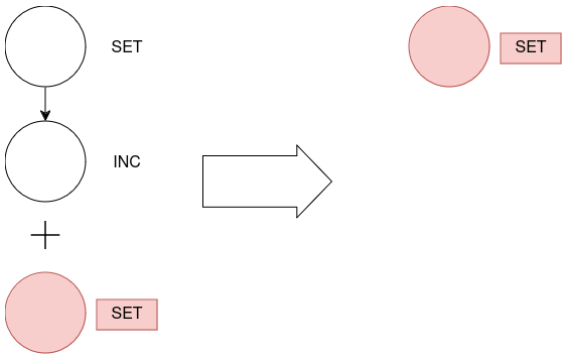


Figure 10: Adding a SET operation on top of an INC operation when the pair (INC, SET) has resolution *take-both-reverse-order* results in the new operation replacing the entire list

As an example, a *SET* followed by an *INCRBY* pair will be of type *take-both-keep-order* because it is ok to keep both in that order, but not the other way around. The operator gives an entry for all pairs that might happen. It is thus important that the operator also gives the mirror entry: a *INCRBY* followed by *SET* is of type *take-both-reverse-order*. If no entry exists for a pair, it is considered as *take-both* by default.

We use these new resolution entries for **appending to an existing list of operations**. The new operation is compared to the last already existing:

- if there is none, the new operation is added
- if there is one, we take both in the (existing, new) order and check the associated relationship

  if it is *take-both*, or *take-both-keep-order*, they are "compatible" together: the new operation is added at the end. See Figure 9

  if it is *take-both-reverse-order* or *take-one*, they are "incompatible" in that order: the list of existing operations is discarded and the new operation replaces them. See Figure 10

We also use these entries for **resolving conflicts with a merge**, by comparing the first operations of the winning revision and each conflicting revision one by one, in that order:

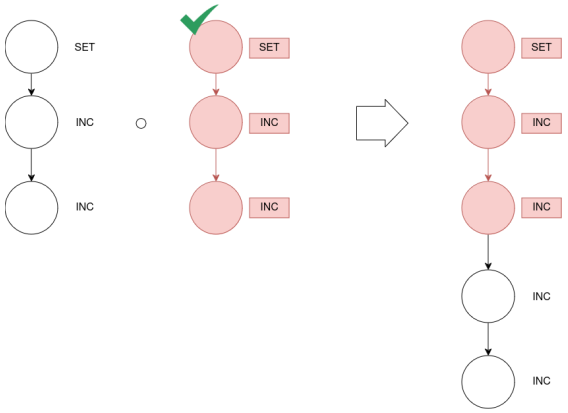


Figure 11: Merging two lists where the first elements are SET and SET, i.e., a *take-one* resolution. The winning version, marked by a green check, is taken

- if it is exactly the same operation (i.e., they have the same identifier), then keep it and use all operations after
- if the result is a *take-one*, then the new list will be the first operation of the winner followed by all subsequent operations in the winner and in the conflict (excluding the first operation in the conflict). See Figure 11
- if the result is a *take-both* or *take-both-keep-order*, both operations are kept, in that order, along with all operations coming after in both versions
- if the result is *take-both-reverse-order*, both operations are kept, in the reverse order, along with all operations coming after in both versions. See Figure 12

We now have a new list that represents the resource and is synchronized on all nodes. The list may or may not contain elements from the previous, pre-merge version.

3.2.4 Running operations

Once the definitive set of operations is defined, Cheops needs to run them. In the list, some operations have already been ran, others are new, i.e., not ran locally yet. There are three cases here:

- If the new operations are strictly at the end, i.e., all operations before have already been ran, then only run the new operations
- If some new operation is in the middle or at the beginning, i.e., some operations after it is already run, then the state is invalid.

  If the very first two operations, in that order, are of resolution type *take-both-keep-order* then the order matters. We must restart from the beginning and re-run all operations. This will be ok because it usually means the very first operation is of type replace, so all state will start from a clean slate

  Otherwise if the same pair is of type *take-both*, so all operations are commutative: running only new operations is ok

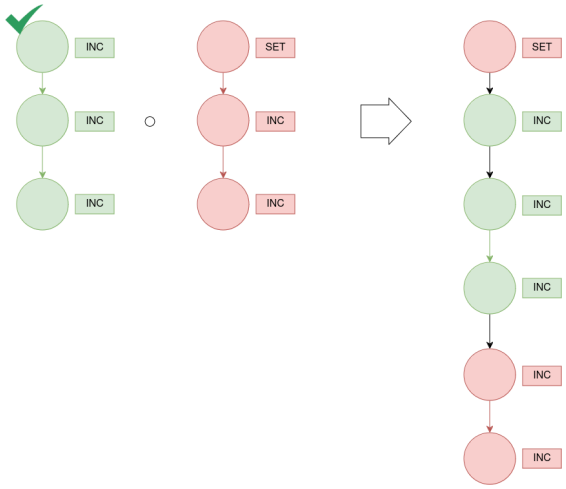


Figure 12: Merging two lists where the first elements are INC then SET (winning always comes first), i.e., a *take-both-reverse-order* resolution. The result takes both and appends the rest

Note that other cases are not possible, because Cheops will have restarted the list of operations before

Once we know which operations we actually need to run, they are passed to the Backend layer in charge of running them all on the underlying system and returning the status (*OK* or *KO*) to the Replication layer. That status is stored in another JSON Document and sent to CouchDB, with the same list of nodes as the resource, to be replicated. Much like the replication of the list of operations, this broadcast ensures that all information eventually spreads back to the user.

3.2.5 Taking a step back

From the user point of view, this is what happens:

1. The user sends an operation to a node they select. This node is thereafter called the *receiving* node, because it received the query, and will be responsible for the request during its entire lifetime
2. The above processes are run, operations are broadcasted, run everywhere, and the replies are also broadcasted
3. The receiving node sees the replies for the operation and give them back to the user. If not all replies are seen before a 30-seconds timeout the initial receiving node sends back a *timeout reply* instead and stops waiting.

This behaviour gives the operator a sense of working in a synchronous way with Cheops. A CLI tool makes all of this operation easier: it will automatically transform all the input into the proper HTTP response and properly show the different replies to the user. It is the preferred way of interacting with Cheops.

Cheops also cares about operational needs. The API, and thus the CLI, have a way for the operator to query the current state of the resource by sending a custom command, directly executed and not registered as an operation. Cheops also provides a way for the operator to

know if there is an error anywhere following the application of an operation, so that they can take appropriate measures. An error will most certainly be followed with the resource diverging so it must be dealt with promptly. When the operator knows what to do to solve it, they can pass a *force* parameter to tell the system it is a proper resolution. The CLI tool can also be used to query the current state of the resource by passing a *show* command that will be directly executed on all nodes and give back the result to the user.

3.3 Areas of improvement

Here are some of the pain points related to how Cheops works, either because of its architecture or because of its first-implementation status, and where future development may propose a boost in performances and usability.

3.3.1 Storing data only once

Imagine an operation saying "insert this image in the filesystem". This operation must maintain the whole image in CouchDB for the future, but the image will also be stored in the filesystem as this is the meaning of the operation. To reduce this duplication, multiple strategies can be used:

if the application stores its data in a deduplicating filesystem, CouchDB can be configured to use the same filesystem (for example a shared ZFS filesystem on a cluster). the other way around: if the application can choose its storage space, CouchDB can serve the file thanks to its HTTP endpoints. The caveat is that CouchDB becomes a strong requirement of Cheops, whereas today it is only an intermediary for sync.

3.3.2 Pruning operations when all nodes agree

The list of locations for a given resource is known and cannot change. Moreover, every node knows the execution status of all operations from other nodes. Thanks to this every node can know when operations have been properly executed on all other nodes: it is thus acceptable, if they have been properly run, to remove them from the list of all operations. Note that while it is correct from the point of view of operations, the resource might still diverge in the application. Imagine an operation that compresses some common file: if the exact compression details are not the same, the result will differ. It is important for operators to keep a history of operations to understand why they end up with different objects. From a first approximation though it is ok to prune operations if they are deterministic.

3.3.3 Hooks

Cheops has an optimistic 30-seconds window during which it hopes to have synchronized and executed the operations on all nodes, to give the operator a feeling of working synchronously, but by the very nature of our work some operation might not have been synchronized to other nodes (because they or the network is down, typically). This doesn't prevent Cheops from working in the background continuously: if the network is back up 24 hours after the operation has been inserted, it will be synchronized, run, and the result will be synchronized back everywhere. There is no way for the operator to know about this: outside of the 30-second window all operations happen in the background. An operator might still be interested in knowing when something happens: either when a specific operation was run, or more generally when a resource has been modified. It is possible to extend Cheops to offer a system of hooks for this. The simplest way is to plug into CouchDB. It has the *changes* endpoint facilitating realtime following of changes (this

is what Cheops itself uses) as described in its documentation. However this again puts CouchDB as a fundamental brick of the solution and prevents any change in that direction. It might be more interesting to offer a simplified changes feed at the Cheops level (maybe /changes/id to follow changes of a specific resource), and tell Cheops to follow changes inside CouchDB for that hypothetical new endpoint. It wouldn't take more than 2 days for an experienced engineer to build this. This solution is simplest but requires the operator's computer to always be turned on: since it is a pull-based mechanism, something needs to continuously (try to) pull. A more involved mechanism would be push-based, controlled by Cheops itself:

- sending an email by connecting to a preconfigured smtp server with a specific account
- sending a message on an irc server
- ping a webhook
- run any kind of command for any scenario (push a log in a supervision system, ...)

Webhooks are often available in current messaging tools and can be a good way to inform a team of operators, where they usually discuss, that something happened on a resource. The potential issue with this approach is that all Cheops node are independent, and those mechanisms will run independently on each node. If the operator wishes to know whenever something happens on any node, that is fine, but if they only want a summary information when all nodes have run the operation some coordination will be required. An experienced engineer wouldn't take more than 3 days to build the first version, but it could take a week to devise and implement a "summary" version where only one action is taken when all the nodes have run the same operation on the same resource.

3.4 Chephren, resources interaction for Cheops

Four students contributed to the Cheops project by building Chepren, a web interface to visualize the state of the system, of the nodes and of the resources. More detailed can be found in [3, 4]. This interface allows users to see at a glance where the resources are situated and whether the state is the one expected. It was done with an earlier version of the API and as such needs to be updated with the latest developments.

As visible in Figure 13 Chephren displays a list of all the nodes in our network, along with a summary count of resources on each of them. Figure 14 shows the list of resources available on each node, along with some other possible actions:

- Every resource can be modified (i.e. a new operation can be run on it) or deleted. Note that deletion hasn't been implemented yet on Cheops for priority reasons
- A new resource can be created
- the node can be "turned off" to simulate disconnections and evaluate how the system behaves when everything is down

Figure 15 shows a parallel view of all the operations for a given resource as seen on all nodes. With this view, an operator is able to quickly see the current synchronization status for every resource and understand why a potential divergence might appear. Figure 16 shows the same resource on the same nodes, but node 3 misses one of the operations.

Chephren was implemented as a stand-alone project: it is a full-stack backend and frontend, and the nodes are fake nodes that can easily be turned off and on. The idea was to allow Chephren to evolve at its own pace and define what APIs were needed for it to work (the list of

Figure 13: Chephren's main screen

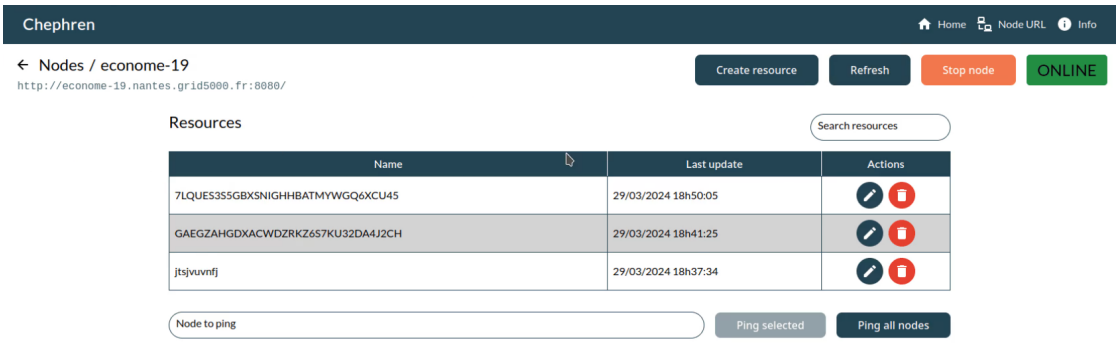


Figure 14: Resource list on a specific node in Chephren

Figure 15: List of operations for a resource as seen on all nodes

Figure 16: Diverging list of operations for the same resource

endpoints, the format they should have, the expected parameters and return data) and to gradually implement those APIs in Cheops. Those APIs have been implemented but unfortunately development of Cheops has had to re-start multiple times from a novel data model, and the current version doesn't support Chephren. A quick back of the envelope calculation estimates it shouldn't take more than a week for a developer experienced in how Cheops works to implement the API chephren needs.

4 Experimentations

Our approach separates the consistency concerns from the underlying application logic. We implement the approach on our **Cheops orchestrator** [11], which geo-distributes an application in a generic and non-intrusive way. We perform our experiments over the replication collaboration that replicates a resource with multiple instances of the same application. We utilize Grid'5000 [1] platform for our experiments.

We perform our experiments to demonstrate three proposals:

- Synchronization of resource: We perform experiments to ensure that operations are performed at each site and concurrent operations are managed by the Cheops orchestrator.
- Network disconnection: We perform experiments by simulating network disconnections and ensuring that once the network is online, all the resources are eventually synchronized
- Handling failure in eventual convergence: After the network disconnection, we simulate a condition in which we handle failure while synchronizing the resource eventually.

4.1 Protocol

Thanks to EnOSlib [7] it was possible to orchestrate Grid5000 to automatically provision the required number of servers, install the required software and run the tests automatically.

Cheops is intended to be used for any application. To test this hypothesis we will do three series of tests with three different applications:

- Kubernetes, where the resource is a deployment
- redis, where the resource is an integer
- the filesystem, where the resource is a file

For all scenarios the tests will be the same and involve multiple conflicting operations on a single resource. The goal of the tests is to run those operations, see how they behave, and hopefully converge on the same state. Our experiment testbed has control over the network and can cut and restore it at any time to simulate disconnections: every time the network is cut, we cut it between all nodes such that no node can talk to any other. After restoring it, the testbed ensures that

- all operations are synchronized
- all operations that need to be run are run
- all replies of all operations that have been run are synchronized on all nodes

[11] https://gitlab.inria.fr/discovery/cheops

For each scenario, we run 3 different independent tests, each on a different resource to simulate a clean reset. The 3 tests are the same are the same for the 3 scenarios. They always start with an initialization operation, and then the rest:

1. In the first test each operation is run sequentially. We wait for the reply of each operation to be synchronized before running the next. There is no concurrency involved here, we merely want to verify operations work

2. In the second test we cut the network and run each conflicting operation on a different node. Network is rested, we wait for the synchronization and inspect the final result: it must be the same on all nodes

3. Finally we do the same as the previous one, except one of the operations is voluntarily failing to simulate a local error. We inspect the end result and expect it to be similar on all nodes

The entirety of the code for creating nodes, installing the necessary building blocks and running tests is included in the Cheops repository[12].

4.2 Kubernetes

The first scenario involves 3 Kubernetes clusters clusters located in *site 1*, *site 2* and *site 3* along with Cheops at each site. The 3 clusters are totally independent and not linked with any existing solution. They don't even know other clusters exist. An internal state of a Kubernetes resource, here a deployment, is described by a definition written in a YAML structure. The basic type of data is a key-value pair in either of these structure. All operations for a key-value pair belongs to *replace operations*. Hence, we construct an *Operations resolution matrix*, as defined in the previous chapter, by identifying the operations for a Kubernetes resource type, as can be seen in Table 4.

Here's the initialization request, showing how the CLI is used

```
$ cheops \
    --id deployment-id \
    --sites 'site1&site2&site3' \
    --command "sudo kubectl create deployment -f recipe.yml" \
    --config config.json \
    --type create
```

It takes multiple arguments: the id of the ressource, the sites definition, the command to execute, the configuration of the resource, and the type of it. The command specifies the application operation (which in the case of kubernetes relies on a resource source file ($-f$)), collaboration type (a replication here because of &), operation type, and *operations resolution matrix* in the config.json file. The recipe here is a standard kubernetes bootcamp deployment, with a single container (containing the image at gcr.io/google-samples/kubernetes-bootcamp) having a single replica. Our tests will play on the number of replicas and check how Cheops and the clusters behave. In practice, our automated tests don't use the CLI but the HTTP API. This has no effect on the results. We will illustrate orders with the cli for readability

The 3 conflicting operations are the following:

- apply 2 replicas on site 1:

[12] https://gitlab.inria.fr/discovery/cheops

```
$ cheops \
    --id deployment-id \
    --command "sudo kubectl apply -f apply-recipe.yml" \
    --sites 'site1&site2&site3' \
    --type apply
```

- patch to have 3 replicas on site 2:

```
$ COMMAND="sudo kubectl patch deployment deployment-id -p '{\"spec\": {\"replicas\": 3}}'"
$ cheops \
    --id deployment-id \
    --command "$COMMAND" \
    --sites 'site1&site2&site3' \
    --type apply
```

- and replace to 4 replicas on site 3:

```
$ cheops \
    --id deployment-id \
    --command "sudo kubectl replace -f replace-recipe.yml" \
    --sites 'site1&site2&site3' \
    --type apply
```

Again, the cli is shown here to easily understand how Cheops is designed, but the HTTP api is used in the automatic tests. Cheops detects conflicts and checks for operation combinations with the *operations resolution matrix* (see Table 4). Since there is only a single type of operation and the relationship is *take-one*, then in case of conflict, only one should remain.

First operation	Second operation	Relationship
Apply	Apply	take-one

Table 4: Operations matrix for a Pod resource

The first test sequentially changes the number of replicas of the pods: this is properly managed and we can see the right number of replicas at each site after each operation. At the end of the operations we have 3 sites each with a deployment containing 4 replicas.

The second test sees the 3 recipes in conflict, yet after the synchronization is over the 3 sites have a deployment with the same number of replicas. It is impossible to predict in advance what that number will be, and in practice multiple runs give a different number. This is expected: Cheops makes sure the state is convergent, but there is no way to determine which version should "win".

The third test was done by replacing the last recipe with one where the number is not an integer: this is a failure for kubernetes. Indeed after the operations were done and synchronization was done, we could see that that operation failed. However since the deployment was already up with one replica at the beginning, the resource was still in a working state. The same reconciliation algorithm happened and selected one of the operations, not known in advance, but deterministically. The end result was always a deployment with the same number of replicas, however since we don't know which one was used in the end, two cases happened:

- in some runs the failing operation won: the number of replicas is the one from the original recipe, i.e 1
- in some runs another, working operation won: that gave the number of replicas

4.3 Redis

The second scenario involves 3 independent redis nodes located on 3 different sites.

The 3 instances are also all independent and not installed in a Cluster or Sentinel mode so as to be unaware of each other. The resource is a counter, for which the *operations resolution matrix* is described in Table 5. The reference of commands can be seen at https://redis.io/docs/commands/. What we can see from this table is that the *INCRBY* operations might be compatible but with the *SET* in the mix, the *SET* will always replace the other operations.

The 3 conflicting operations are the following:

- set a value on site 1:

```
$ cheops \
    --id resource-id \
    --command "redis-cli set resource-id 29" \
    --sites 'site1&site2&site3' \
    --type set
```

- increase by 13 on site 2:

```
$ cheops \
    --id resource-id \
    --command "redis-cli incrby resource-id 13" \
    --sites 'site1&site2&site3' \
    --type inc
```

- and increase by -12 on site 3:

```
$ cheops \
    --id resource-id \
    --command "redis-clie incrby resource-id -12" \
    --sites 'site1&site2&site3' \
    --type inc
```

Note that 2 of the operations are not conflicting: *INCRBY* are commutative with each other, and this is seen in the matrix. What we wanted to see in this test is the interaction with the *SET* command.

The first test runs operations sequentially, and it is easy to see that the resource should have a value of 30: this is what we observe in all our runs

The second test sees the 3 operations interact according to Table 5 and, as we have seen above, the expected result should be as if there only was the *SET* operation, and the value should be 29: this is what we observe.

The third test replaces one of the *INCRBY* with a *SADD* command, still with the same type: *inc*. This command adds an object to a set which is why we keep the same type. Since it only works on set objects and our resource is an integer, it is expected to fail and this is what happens. However since it is still an *inc* operation, it follows the same fate as the previous *INCRBY* operation and is superseded by the *SET* operation: the result should be exactly the same. This is what we observe in our runs.

First operation	Second operation	Relationship
Set	Inc	take-both-keep-order
Inc	Set	take-both-reverse-order
Inc	Inc	take-both

Table 5: Operations matrix for an integer resource in redis

4.4 Filesystem

The last scenario is the one that was used for developing: we used the filesystem as an application, and a specific file as a resource. The file is initialized with a specific content. The 3 operations are the following:

- append content on site 1:

```
$ cheops \
    --id filename \
    --command "echo left >> filename" \
    --sites 'site1&site2&site3' \
    --type add
```

- replace content on site 2:

```
$ cheops \
    --id filename \
    --command "echo middle > filename" \
    --sites 'site1&site2&site3' \
    --type set
```

- append content on site 3:

```
$ cheops \
    --id filename \
    --command "echo right >> filename" \
    --sites 'site1&site2&site3' \
    --type add
```

Because of these operations the initial content should never be found as-is after the execution. As we can see we have two operations appending content at the end of the file, and one replacing the full content. The operations resolution matrix in Table 6 shows that *set* and *add* pairs will see *set* win, but there is nothing for two *add* operations: in fact, this pair is a **Class 3** pair and as such isn't handled by Cheops yet. In this specific case we didn't set any resolution mechanism in Cheops. When this happens, Cheops defaults to a **Class 1** pair, i.e two commutative operations. This is obviously wrong in our example but it is fine because in any case we expect the *set* operation will win and override all other operations.

The failing test was done by replacing the append operation by the following:

First operation	Second operation	Relationship
Set	Add	take-both-keep-order
Add	Set	take-both-reverse-order
Set	Set	take-one

Table 6: Operations matrix for a file

```
$ echo right >> filename/subfile
```

which is invalid because a file doesn't have subfiles. The situation is the same as the one in redis, and the set operation won on all sites: the state eventually converged.

Appendix

Code

The code for Cheops is available here: https://gitlab.inria.fr/discovery/cheops/-/tree/main; the code for Chephren is available here: https://gitlab.imt-atlantique.fr/chephren/chephren.

Cheops communication

Conferences

ICFEC 2024 Geo Johns Antony, Marie Delavergne, Adrien Lebre, Matthieu Rakotojaona Rainimangavelo. Thinking out of replication for geo-distributing applications: the sharding case. ICFEC 2024 - 8th IEEE International Conference on Fog and Edge Computing, May 2024, Philadelphia, United States. pp.1-8. ⟨hal-04522961⟩

ICSOC 2022 Delavergne, M., Antony, G.J., Lebre, A. (2022). Cheops, a Service to Blow Away Cloud Applications to the Edge. In: Troya, J., Medjahed, B., Piattini, M., Yao, L., Fernández, P., Ruiz-Cortés, A. (eds) Service-Oriented Computing. ICSOC 2022. Lecture Notes in Computer Science, vol 13740. Springer, Cham., doi: 10.1007/978-3-031-20984-0_37.

Euro-Par 2021 Cherrueau, RA., Delavergne, M., Lebre, A. (2021). Geo-distribute Cloud Applications at the Edge. In: Sousa, L., Roma, N., Tomás, P. (eds) Euro-Par 2021: Parallel Processing. Euro-Par 2021. Lecture Notes in Computer Science(), vol 12820. Springer, Cham., doi: 10.1007/978-3-030-85665-6_19.

Workshops

XP 2021 Workshops Delavergne, M., Cherrueau, RA., Lebre, A. (2021). A Service Mesh for Collaboration Between Geo-Distributed Services: The Replication Case. In: Gregory, P., Kruchten, P. (eds) Agile Processes in Software Engineering and Extreme Programming –

Workshops. XP 2021. Lecture Notes in Business Information Processing, vol 426. Springer, Cham. doi: 10.1007/978-3-030-88583-0_17.

Research reports

RR 2022 Marie Delavergne, Geo Johns Antony, Adrien Lebre. Cheops, a service to blow away Cloud applications to the Edge. [Research Report] RR-9486, Inria Rennes - Bretagne Atlantique. 2022, pp.1-16. ⟨hal-03770492v2⟩.

RR 2020 Ronan-Alexandre Cherrueau, Marie Delavergne, Adrien Lebre, Javier Rojas Balderrama, Matthieu Simonin. Edge Computing Resource Management System: Two Years Later!. [Research Report] RR-9336, Inria Rennes Bretagne Atlantique. 2020. ⟨hal-02527366v2⟩.

Other content

Marie Thesis Marie Delavergne. Cheops, a service-mesh to geo-distribute micro-service applications at the Edge. Distributed, Parallel, and Cluster Computing [cs.DC]. Ecole nationale supérieure Mines-Télécom Atlantique, 2023. English. ⟨NNT : 2023IMTA0347⟩. ⟨tel-04081084⟩

OpenInfra Summit Geo Johns Antony, Marie Delavergne and Baptiste Jonglez. Cheops - Can a "service mesh" be the right solution for the Edge?, https://www.youtube.com/watch?v=7EZ63DMRJhc, OpenInfra Summit 2022, Berlin - Accessed: 2023-01-05.

References

[1] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[2] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[3] Titouan Bizet, Aymeric Lecolazet, Bruno Preka, Julien Reig, and Matthieu Rakotojaona-Rainimangavelo. Chephren. https://gitlab.imt-atlantique.fr/chephren/chephren. Accessed: 08/2024.

[4] Titouan Bizet, Aymeric Lecolazet, Bruno Preka, Julien Reig, and Matthieu Rakotojaona-Rainimangavelo. Chephren: Simplifier l'interaction avec des ressources distribuées. https://gitlab.inria.fr/discovery/cheops/-/blob/main/infos/Poster_Cheops_Chephren-8.pdf. Accessed: 08/2024.

[5] Ronan-Alexandre Cherrueau, Javier Rojas Balderrama, and Matthieu Simonin. OpenStackoïd: Make your OpenStacks Collaborative. https://gitlab.inria.fr/discovery/openstackoid. Accessed: 02/2020.

[6] Ronan-Alexandre Cherrueau, Marie Delavergne, and Adrien Lebre. Geo-distribute cloud applications at the edge. In *EURO-PAR 2021-27th International European Conference on Parallel and Distributed Computing*, 2021.

[7] Ronan-Alexandre Cherrueau, Marie Delavergne, Alexandre van Kempen, Adrien Lebre, Dimitri Pertin, Javier Rojas Balderrama, Anthony Simonet, and Matthieu Simonin. EnosLib: A Library for Experiment-Driven Research in Distributed Computing. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1464–1477, June 2022.

[8] Ronan-Alexandre Cherrueau, Adrien Lebre, and Javier Rojas Balderrama. Implementing localization into openstack cli for a free collaboration of edge openstack clouds. OpenStack Summit, Berlin, Canada, https://www.openstack.org/videos/summits/denver-2019/implementing-localization-into-openstack-cli-for-a-free-collaboration-of-edge-openstack-clouds, December 2019.

[9] CouchDB. Apache CouchDB. https://couchdb.apache.org/. Accessed: 08/2024.

[10] Marie Delavergne. *Cheops, a service-mesh to geo-distribute micro-service applications at the Edge*. Theses, Ecole nationale supérieure Mines-Télécom Atlantique, March 2023.

[11] Marie Delavergne, Geo Johns Antony, and Adrien Lebre. Cheops, a service to blow away cloud applications to the edge. In *International Conference on Service-Oriented Computing*, pages 530–539. Springer, 2022.

[12] W. Li et al. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.

[13] Renaud Lottiaux et al. Openmosix, openssi and kerrighed: a comparative study. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 1016–1023. IEEE, 2005.

[14] Tuyen X Tran, Abolfazl Hajisami, Parul Pandey, and Dario Pompili. Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges. *IEEE Communications Magazine*, 55(4):54–61, 2017.

RESEARCH CENTRE
Centre Inria de l'Université de Rennes

Campus universitaire de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803