Cheops reloaded, further steps in decoupling geo-distribution from application business logic : a focus on externalised sharding collaboration, consistency and dependency

Geo Johns Antony

▶ To cite this version:

Geo Johns Antony. Cheops reloaded, further steps in decoupling geo-distribution from application business logic : a focus on externalised sharding collaboration, consistency and dependency. Distributed, Parallel, and Cluster Computing [cs.DC]. Ecole nationale supérieure Mines-Télécom Atlantique, 2024. English. NNT : 2024IMTA0441 . tel-04917025

HAL Id: tel-04917025
https://theses.hal.science/tel-04917025v1

Submitted on 28 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

COLLEGES BRETAGNE LOIRE | SCIENCES POUR L'INGENIEUR ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS DE LA LOIRE – IMT ATLANTIQUE

ÉCOLE DOCTORALE Nº 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Informatique*

Par
Geo Johns ANTONY

Cheops reloaded : Further steps in Decoupling Geo-Distribution from Application business logic

A focus on externalised Sharding collaboration, Consistency and Dependency

Thèse présentée et soutenue à IMT Atlantique, Nantes, le 17 decembre 2024
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)
Thèse Nº : 2024IMTA0441

Rapporteurs avant soutenance :

Fabienne Boyer	Professeure, Université Grenoble Alpes
Pierre Sens	Professeur, Sorbonne Université

Composition du Jury :

Président :	Dalila Tamzalit	Professeure, Université de Nantes
Examinateurs :	Fabienne Boyer	Professeure, Université Grenoble Alpes
	Pierre Sens	Professeur, Sorbonne Université
	Fabien Baligand	Ingénieur de recherche, CEA
Dir. de thèse :	Adrien Lebre	Professeur IMT Atlantique (actuellement en detachement DR Inria)

ACKNOWLEDGEMENT

This doctoral thesis represents the culmination of many years of learning, questioning, refining, and growing—both intellectually and personally. It would not have been possible without the dedicated support and guidance of numerous individuals and institutions, to whom I owe my deepest gratitude.

First and foremost, I would like to thank my supervisor, Adrien Lebre, whose patience, encouragement, and scholarly insight have been a beacon throughout this process. He pushed me to explore deep into the unknown. His trust in my ideas, willingness to engage critically with my work, and his constructive feedback have shaped my development as a researcher and thinker. Along with my supervisor, I would like to thank my colleague, Marie Delavergne, who was like a second supervisor to me. She guided me through the entire journey and provided strong support, both intellectually and personally.

I would equally like to thank my defense jury: Dalila Tamzalit, Fabienne Boyer, Pierre Sens and Fabien Baligand, who each offered invaluable counsel, pointed questions, and thoughtful critiques.

There were many people who contributed to the development of my thesis, and I would like to highlight a few in particular. I extend my gratitude to Matthieu Rakotojaona Rainimangavelo, for his contributions to the projects development and for his invaluable insights and discussions, especially regarding consistency approaches that enriched my work. I am also grateful to Eloi Perdereau for his resource and application related insights, and valuable feedback throughout my journey. Finally, I would like to thank Baptiste Jonglez, who engaged in countless discussions about the Cheops architecture and provided constructive feedback that helped improve this thesis.

I want to extend my appreciation to everyone in the STACK research team, who fostered a supportive, friendly environment throughout this long process. While I am not naming each of you individually, please know that your encouragement and feedbacks meant the world to me.

I would like to thank Anne-Claire Binetruy and Catherine Fourny for their administrative assistance. Their help navigating the relationships between Inria and IMT Atlantique eased many of the challenges I faced along the way.

Finally, I want to express my deepest gratitude to my family: Antony M John, Jessy George, and Jelena Antony, for your love and support. I am also immensely grateful to Celia Kessassi, who stood by me throughout the entire journey, serving as a constant source of strength and a go-to person. To my friends, who were with me throughout this journey, especially Melvin Tomy, Vishal Issac, and Bimal Narayanan, for your unwavering support, particularly during the final moments of my defense.

Table of Contents

List of figures 8

List of tables 10

Introduction 12

I Context 19

1 From Cloud to Edge application **21**
- 1.1 Cloud computing . . . 21
  - 1.1.1 Cloud resource . . . 23
  - 1.1.2 Cloud Application . . . 24
  - 1.1.3 Service Oriented Architecture . . . 25
- 1.2 Edge Computing . . . 25
  - 1.2.1 Challenges with Edge . . . 26
- 1.3 Shift from Cloud to Edge Computing . . . 27

2 From distributed to geo-distributed application **29**
- 2.1 Distributed Cloud Application . . . 29
- 2.2 From distributed to geo-distributed applications . . . 30
  - 2.2.1 Inter-Service collaboration . . . 31
  - 2.2.2 Broker based collaboration . . . 32
  - 2.2.3 Database collaboration . . . 33
  - 2.2.4 Limitations with existing approaches . . . 34
- 2.3 Summary . . . 35

3 A solution to externalize geo-distribution **37**
- 3.1 Creating a generic and non-intrusive solution . . . 37
  - 3.1.1 Collaborations . . . 41
  - 3.1.2 An updated Cheops principles . . . 43

3.2 Limitations with our existing approach 44
3.2.1 Issues with replication in general 44
3.2.2 RAFT based consensus may not be enough 46
3.2.3 Difficulty in creating an illusion of a single application 47
3.3 Research Questions 48
3.4 Summary 48

II State of the art 51

4 Existing geo-distribution solutions 53
4.1 Design for our approach: Research Question 1 53
4.2 Comparison Points 54
4.3 Existing solutions Collaboration 56
4.4 Summary 66

5 Consistency approaches 69
5.1 Design for our approach: Research Question 2 69
5.2 Comparison Points 69
5.3 Existing solutions for Consistency 71
5.4 Summary 82

III Contributions 83

6 New Cheops Architecture 85
6.1 Existing Architecture 85
6.1.1 Limitations with the Existing architecture 88
6.2 New Architecture 89
6.3 Experimental setup 92
6.4 Summary 93

7 Thinking beyond replication, an approach to geo-distribute an application with sharding 95
7.1 Creating a collaboration with Sharding 95
7.1.1 Sharding vs Replicating a resource 97
7.1.2 Challenges with existing sharding approaches 98

7.2 Cross Collaboration . . . 98
7.2.1 Extension: Generating new shards . . . 99
7.2.2 Aggregation: a consolidated view of the shards . . . 107
7.3 Can a resource be completely sharded? . . . 111
7.4 Validation . . . 112
7.4.1 Use-case: Extending Cross to Sharelatex . . . 116
7.5 Summary . . . 117

8 Externalizing consistency between geo-distributed Instances 119
8.1 An External Approach Towards Consistency . . . 119
8.2 Extending stateful operations . . . 122
8.3 Classifying Operations . . . 125
8.3.1 Class 1: Iterative & commutative operations . . . 126
8.3.2 Class 2: Replace & non-commutative operations . . . 126
8.3.3 Class 3: Iterative & non-commutative operations . . . 128
8.4 How can Cheops map a class to an operation? . . . 133
8.5 Handling exceptions if eventual convergence fails . . . 136
8.6 Validation . . . 139
8.6.1 Kubernetes Pods . . . 140
8.6.2 Kubernetes Deployments . . . 141
8.7 Summary . . . 142

9 Handling Dependencies externally for geo-distributed instances 145
9.1 Managing Relationships . . . 145
9.2 Relationship model . . . 146
9.2.1 Requires Relationship . . . 148
9.2.2 Reliance & non-Transitive Relationship . . . 148
9.2.3 Reliance & Transitive Relationship . . . 149
9.2.4 Local Relationship . . . 150
9.2.5 Cascading relations . . . 152
9.3 How can Cheops identify these relationships? . . . 152
9.3.1 Matrix based solution for handling dependencies . . . 153
9.3.2 Relationship logic . . . 158
9.3.3 A workflow to ensure an illusion of a single application . . . 163
9.3.4 Limitation . . . 165

9.4 Validation 165
9.5 Summary 169

Conclusion and Perspectives **171**

Résumé en français **179**

Bibliography **187**

List of Figures

1.1 Example services available to a Cloud Consumer by NIST [26] 22
1.2 Resource creation in cloud application 24
1.3 Cloud, Edge and Fog computing overview, cited from [45] 26

2.1 Existing approaches for Collaboration and their issues 31
2.2 A basic Broker based design 35

3.1 A cloud application *App* with two services *a* and *b* along with a user (represented with a black dot) initiating the request 37
3.2 Two instances *App* with two services *a* and *b* deployed on two sites, where service *a* from *site* 1 depends on *b* from *site* 2 38
3.3 Two instances of a cloud application *App*; user makes a request k on Site 1, it is forwarded to the remote instance with Cheops 39
3.4 A single site (unified) abstract view created from two instances of a cloud application with Cheops 40
3.5 Two instances of a cloud application *App*; user makes a request k on site 1, it is forwarded to the remote instance with Cheops 41
3.6 Two instances of a cloud application *App*; user makes a request k on site 1, it is forwarded to the remote instance with Cheops 43
3.7 Single site cloud application portraying :(a) a strong relation between secret and pod (b) replicating the pod in the same site 47
3.8 The illusion provided by Cheops fails as the pod fails when trying to geo-distribute it to Site 2, due to the absence of dependent secret resource . . . 47

4.1 Liqo control plane from [63] 58

6.1 Basic Cheops architecture 85
6.2 Existing Cheops architecture 86
6.3 New Cheops architecture 89
6.4 New Cheops CLI 90

6.5 Experimental setup with Cheops and Kubernetes 92

7.1 An example of a tabular data divided into shards 96
7.2 (a) Replicating and (b) Sharding a resource 97
7.3 Scope-lang expression syntax for Cross collaboration model 104
7.4 Extension workflow in Cross collaboration model 106
7.5 Aggregation response in Cross collaboration model 108
7.6 Partial error mechanism for Cross collaboration model 109
7.7 Cross Namespace: Extended & Aggregated View 113

8.1 Replace & Iterative: solution 1 127
8.2 Replace & Iterative: solution 2 128

9.1 Cheops relationship model 147
9.2 Sample configuration file for: (a) Heat orchestration template for OpenStack (b) Deployment file for Kubernetes 157
9.3 Cheops components 172
9.4 Different logics in Cheops 173

List of Tables

4.1 Existing Collaboration approaches to geo-distribute an application 66

5.1 Existing Consistency approaches to geo-distribute an application 80

8.1 Class representation of concurrent operations and their conflict resolution strategies. 125
8.2 Operation matrix for P-N Counter 134
8.3 Operation matrix for a String 134
8.4 Operations matrix for a Pod resource 135
8.5 Consistency matrix for OpenStack operations on a VM 135
8.6 Consistency matrix for Kubernetes Deployments 141

9.1 Relationship matrix for Kubernetes application 155
9.2 Relationship matrix for Open stack application 155

INTRODUCTION

The cloud computing paradigm has been a fundamental shift in how computational resources and services are delivered [1, 2]. It enables users to access computing resources, including infrastructures and software, on-demand, without the need for direct management of them.

This paradigm relies on a centralized location dedicated to host all the hardware and computing called Data Centers (DC).

The current ongoing shift to Industry 4.0 like applications, such as the Internet of Things (IoT), Artificial Intelligence (AI), autonomous driving, Augmented and Virtual Reality (AR/VR), requires real-time data processing and decision-making closer to the data source. This led to the increasing demand for computing closer to source of the data or proximity to the user, known nowadays as the edge paradigm[3, 4].

Industry 4.0 has not only accelerated the shift from centralized cloud computing to edge devices but also highlighted the need for seamless collaboration between these distributed computing environments. In particular to cope with the geographical dispersion of compute across the globe.

This shift towards geo-distribution, where applications could be spread across geographical regions, has become critical to ensure low-latency, availability, and resilience [5, 6, 7]. There is a substantial amount of research being conducted on this topic, particularly by the STACK research group, where I developed the activities presented in this manuscript.

In most cases, existing geo-distribution solutions embed the distribution logic directly within the application's codebase, meaning the application itself manages how resources are distributed and synchronized across different locations. As a result, if geo-distribution was not considered during the application's initial design, retrofitting a legacy application for geo-distribution becomes a tedious and costly process, requiring developers to make intrusive modifications to the core application code. This approach introduces unnecessary complexity, as it forces developers to alter the fundamental structure of the application.

For instance, federating a service like Keystone across OpenStack application instances, known as Keystone Federation [8], necessitates modifying the existing service to

enable geo-distribution. Similarly, middleware approaches such as Kube-Edge [9] require the creation of specific brokers capable of interpreting the application in order to geo-distribute a Kubernetes application across instances. Additionally, database approaches like AntidoteDB [10] demand substantial modifications to the application code itself to support geo-distribution.

As a result, it becomes challenging to adapt or extend a system to new environments or requirements, as every change may require altering the application internal code [11].

Another issue with these solutions is that they are not generic enough and are often tailored to a specific application, making them difficult to apply across different systems. These solutions usually rely on custom logic or APIs that work only for a particular environment, limiting their adaptability, such as KubeEdge [9], it is designed specific for the Kubernetes application.

This lack of generality means that each application needs its own geo-distribution solution, again increasing development complexity and cost. Additionally, non-generic solutions require significant effort to modify when the application architecture changes or when scaling to new regions, as they cannot be easily reused across different platforms or services.

This analysis led to the research question that is currently being explored within the STACK team:

How can we geo-distribute any application without being intrusive to the existing code base?

My colleague, Marie Delavergne, gave a first response to this question in her PhD thesis [12] and introduced the initial version of the **Cheops** solution.

Cheops: an approach to geo-distribute an application externally

As introduced, the existing solutions for geo-distributing an application is either too specific or require code modifications to the business logic.

To address these issues, the Cheops framework have been proposed by the STACK research group [11, 12, 13, 14]. The solution I outline introduces the first fundamental concept of Cheops.

This framework allows any applications to be geo-distributed without requiring significant changes to their business logic. By utilizing a widely-adopted protocol like REST API for communication, Cheops maintains its generic nature.

Cheops considers a set of independent instances of an application (one per site) and

orchestrates requests according to the need of the user. It operates on two primary principles: **local-first** and **collaborative then**. Local-first ensures that any operation at an application instance functions autonomously. It allows for geo-distributing applications by deploying a complete instance at each site.

However, when resources are not available locally, Cheops facilitates collaboration between independent instances to share resources across sites. Thus the second principle collaborative-then was introduced, where instances collaborate when required. This principle helps applications overcome the limitation of a resource not being available at a local site, by dynamically forwarding and sharing requests between instances.

Current version of Cheops consists of two collaboration mechanisms: Sharing and Replication. Sharing allows independent application instances to dynamically forward requests to the other when a required resource is unavailable locally, enabling effective distribution without duplicating them unnecessarily. For instance, if the compute service *nova* in OpenStack at one site needs an image resource available on the storage service *glance* from another instance, Cheops can forward a request to the remote site to retrieve the resource, allowing the local request to be fulfilled.

Replication, on the other hand, creates and manages identical copies of a resource across multiple instances to ensure consistency and availability, even in the case of network partitions or site failures. This collaboration ensures that critical resources are always accessible across all sites.

An operation can be performed on any of these identical copies from any instance of an application, following the local-first principle. When a change is made to one replica, it must be applied to all other replicas to keep the resource consistent. However, if two users make changes at the same time from different locations to the same replicated resource (called concurrent operations), it can lead to inconsistencies across locations.

Cheops employed a RAFT-like consensus protocol [12] that achieved a strong eventual consistency. Unlike the conventional RAFT protocol [15], this approach initially executes an operation at a local site, upon receiving a request from the user, before waiting to get a majority (quorum) to achieve consensus.

Afterwards, it checks if an operation can achieve the required quorum. If the operation fails to achieve a quorum, a rollback mechanism is in place to revert operations when consensus is not achieved, ensuring that each replica will be eventually consistent. This approach ensures synchronization across geo-distributed sites while preserving the local-first nature of each operation. However, it requires Cheops to store the state of a resource

before applying any operation, enabling a roll-back mechanism if needed.

The first Cheops version, integrates Scope-lang, a Domain Specific Language (DSL) that enables users to specify the geo-distribution requirements alongside their API requests. This language allows the system to understand how to manage and geo-distribute resources across multiple sites based on the user input. As a result, Cheops can function with the native application API along with the added scope-land with each request, making it generic to any.

Through these principles, Cheops creates the illusion of a single, unified system by connecting independent instances of applications (similar to the early single system image (SSI) research [16]), enabling seamless collaboration and resource sharing across geo-distributed environments.

Limitations with the existing approach

While the existing approach offer an initial solution for geo-distributing applications by separating the geo-distribution from the business logic, they still come with certain limitations. Here, I outline these limitations of the current Cheops approach:

1. **Issues with Replication**: Replication, while commonly used (e.g., in CDNs and databases), poses challenges such as high synchronization overhead caused by necessity to perform all the operations at each site. These issues can lead to network congestion (especially at the Edge, due to constraints), delays and inefficient resource usage in geo-distributed environments.

2. **Limitations of RAFT-based Consensus**: Cheops used a RAFT-like consensus mechanism, but it struggled during a network partition, requiring more time to reach quorum, which could lead to multiple operations rolling back. The rollback process itself is complicated, as it requires Cheops to store the state of a resource before each operation. This requires Cheops to be aware about the application context and operations, required to perform the rollback.

3. **Difficulty in Creating the Illusion of a Single Application**: Cheops aims to create the illusion of a unified application running smoothly across geo-distributed instances, allowing any operation that works in a single instance to also work in a Cheops environment. However, problems can occur when a dependency between resources isn't captured. If the dependency isn't available at the remote site, the

geo-distributed resource may fail. This issue is less likely in a local instance, where dependencies are usually present with the resource. This difference in behavior can break the illusion of a unified application across instances.

Research Questions

The highlighted limitations above, motivated us to identify three research questions that led to my thesis:

1. Is it possible to design an alternative collaboration method that addresses replication challenges while adhering to the Cheops principles of prioritizing local-first, non-intrusive, and application-agnostic principles?

2. Can we conceptualize an approach for consistency that overcomes the limitations of the existing Cheops approach, while aligning with its principles of being local-first, non-intrusive, and applicable to any application?

3. Can we ensure that operations within a single instance can be successfully replicated in a Cheops geo-distributed application environment by resolving all dependency issues, thereby maintaining the seamless illusion of a unified application?

Contributions

We published one **research article** and another short-paper to international conferences, as part of my contributions towards the this PhD.

1. The research article was presented at the ICFEC 2024 conference [17], that detailed a novel collaboration model called *Cross*.

2. A short paper was published in ICSOC 2022 conference [13], that detailed the first Cheops architecture and presented an initial outline of my research work.

In addition, I gave several **presentations** towards my research work at various venues:

1. A poster at the Cloud Control Workshop 2024 [18].

2. A poster and gave an oral presentation at the Compas 2022 conference [19].

3. A presentation at the OpenInfra Summit 2022 [20].

4. A presentation at the Journées Cloud 2021 conference [21]

5. An oral and poster presentation at a winter (6th edition of the winter school on distributed systems and networks, 2022) [22] and summer school (First Summer School on Distributed and Replicated Environments (DARE 2023)) [23].

Manuscript Structure

This thesis is structured into three parts with nine chapters:

Part one, explains the context to my research.

In Chapter 1, I explain about cloud applications, the transition to edge and their limitations.

In Chapter 2, I explain the transition from distributed to geo-distributed systems and explain a generic view of the existing solutions with an example for each. I also highlight their limitations an the requirement for a generic and non-intrusive approach.

In Chapter 3, I explain the existing Cheops approach, with an explanation to each of the components. I further explain their limitations and proceed to explain the reason for my thesis by defining the research questions in detail.

Part two, describes the existing relevant solutions, trying to address our research questions and I create a state-of-the-art survey based on it.

In Chapter 4, I explain solution trying to address the first research question in detail and present comparison against our envisioned approach.

In Chapter 5, I explain solution trying to address the second research question in detail and present comparison against our envisioned approach.

State of the art survey regarding the third question is explained in Chapter 9.

Part three, describes my approach to solve all the three research questions.

In Chapter 6, I explain the new architecture for Cheops and the experimental setup we use to validate all of our contributions in this thesis.

In Chapter 7, I explain my approach for the first research question, entitled *Cross*, it allows to shard any resource in an application to geo-distribute them across instances.

In Chapter 8, I explain my approach for the second research question, it presents a method to ensure resources in any applications are consistent across the geo-distributed instances in a generic and non-intrusive manner.

In Chapter 9, I explain my approach to addressing the third research question. This chapter presents a method to ensure the illusion of a single instance is maintained in Cheops by identifying any dependencies in a local instance of an application and resolving it at the remote site.

I further conclude my manuscript with a conclusion and future work that could further enhance the Cheops approach.

Part I

Context

The initial part of my PhD thesis establishes the foundational context for my research, which is centered on geo-distribution of applications across cloud and edge systems.

In Chapter 1, I introduce the paradigms of cloud and edge computing, providing an overview of its resources. This chapter also explores the necessity of edge computing and the challenges associated with it, along with introducing the current trend towards a shift from cloud to edge computing.

With Chapter 2, I delve into the transition from distributed to geo-distributed computing within the cloud and edge paradigms. It offers a detailed discussion of existing methodologies for geo-distributing applications, with an example solution. Additionally, this chapter identifies a set of limitations in these approaches, laying the groundwork for further investigation.

In Chapter 3, I introduce an existing solution developed by the STACK research group to address the first set of limitations. This chapter details each component of this solution and discusses the foundational principles guiding its design. I also address the limitation to the existing approach and outline the key research questions that drive my thesis.

Chapter 1

From Cloud to Edge application

This chapter focuses on presenting a basic idea about cloud and edge computing. It introduces cloud application, resources and the service oriented architecture. Later, I focus on the need for an edge and their limitations. I conclude by highlighting the current shift from cloud applications to edge.

1.1 Cloud computing

Cloud computing provides services such as storage or software (e.g., Google, facebook, etc.), on someone else's powerful computers (called servers) over the internet, instead of having to buy and maintain those hardware on our own. It is like renting space in a giant online warehouse called data centers (D.C), where we can store our files, run programs, and even host websites, without needing to worry about the technical details or costs of owning and managing the hardware, relying less on managing the physical components. This approach is convenient because a large amount of consumers can access these services from anywhere with an internet connection and only pay for what they use. Softwares are designed for cloud, to quickly scale their operations, as they can easily add or remove resources based on demand, making it a flexible and cost-effective solution [1, 2, 24].

These Cloud resources are often packaged together as a model, based on consumers. Each model offer a wide range of options as illustrated in Figure 1.1. There exist many models, as mentioned in the article [25], but for my thesis, I follow the National Institute of Standards and Technology (NIST) [26] definition, it categorizes them into three:

Infrastructure as a Service (IaaS): This category provides the basic building block of computing, like virtual machines (VM), storage, and networking. With IaaS, a consumer can rent these resources to build their own IT infrastructure, just like they would with a physical hardware. It's flexible and scalable, making it ideal for businesses that need to manage and control their computing resources without investing in physical hardware. Examples include, Amazon Web Services (AWS) EC2 and Microsoft Azure compute.

Platform as a Service (PaaS): PaaS offers a platform where developers can build, test, and deploy applications without worrying about the underlying infrastructure. It provides a ready-to-use environment with tools, databases, and operating systems. This service is great for developers who want to focus on coding without handling hardware or software management. Examples include, Google App Engine and Microsoft Azure App Services.

Software as a Service (SaaS): SaaS delivers fully functional software applications over the internet. Users can access these applications through a web browser without needing to install or manage the software on their devices. It's perfect for businesses and individuals who want easy access to software without the hassle of updates or maintenance. Examples include, Google Workspace, Microsoft 365, and Salesforce.

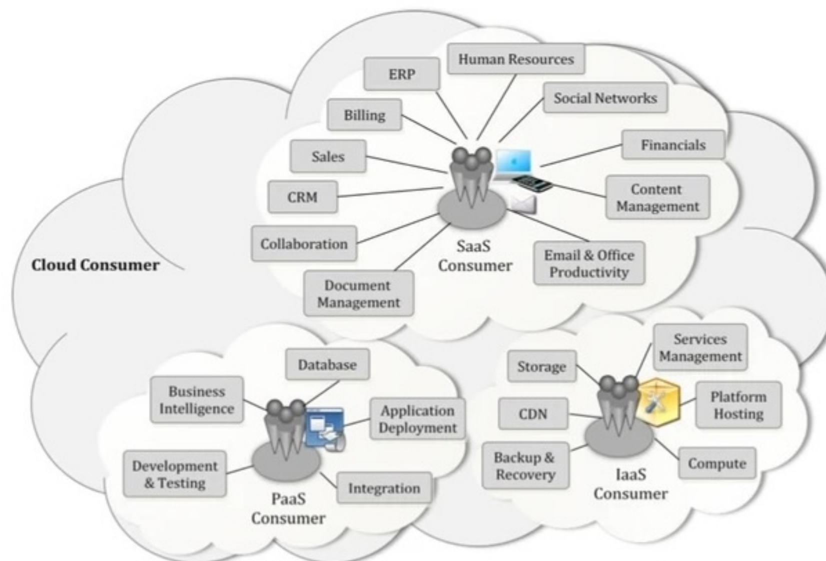

Figure 1.1: Example services available to a Cloud Consumer by NIST [26]

Our research focus is on the software and its distribution across geographically separated locations. Each of these models can be deployed on any Cloud environment. These deployment environments are classified into four, it exist to meet varying needs for control, and flexibility as mentioned by NIST.

A *Private Cloud* is a dedicated environment exclusively for one organization where sensitive information needs to be tightly controlled and stored securely such as government.

A *community cloud* is a infrastructure that is similar to private cloud, it is shared by several organizations with similar requirements and they collectively own, manage and

use the community cloud. They include Microsoft Azure Government Cloud and AWS GovCloud (US), created specific for government requirements.

A *Public Cloud* is a shared environment where multiple organizations or individual users can rent computing resources from a cloud service provider like Amazon Web Services (AWS) or Microsoft Azure to scale their workloads.

A *hybrid cloud* combines elements of both private and public cloud. It allows organizations to keep critical workloads and sensitive data on a private cloud while leveraging the public cloud for less sensitive operations or to handle peak workloads.

1.1.1 Cloud resource

Each of the services illustrated in Figure 1.1 requires different cloud resources to function. Cloud computing offers a variety of resources, which can be broadly categorized (taken from various articles [27, 28, 29]) into the following types:

Compute resources are the processing power provided by cloud, typically in the form of virtual machines (VM) and containers.

Cloud storage resources allow organizations to store, manage, and access data over the internet. These resources are scalable and accessible from anywhere. They typically include object storage such as Amazon S3, Google Cloud storage, etc., block storage such as AWS EBS (Elastic Block Store), Azure Disk Storage, etc., file storage such as Amazon EFS (Elastic File System), Google Filestore, etc.

Cloud networking resources enable the connection and communication between cloud services and users across the internet. They typically include Virtual Private Cloud (VPC) such as AWS VPC, Google VPC, etc., load balancers such as Elastic Load Balancing, Nginx, etc., Content Delivery Networks (CDNs) such as Amazon CloudFront, Azure CDN, etc.

Cloud databases provide managed database services that can scale as required. This service eliminate the need for users to manage database infrastructure. They typically include relational databases such as Amazon RDS (Relational Database Service), Oracle MySQL server, NoSQL databases such as MongoDB atlas, Google Cloud Firestore, etc., and data warehousing such as Amazon Redshift, Google BigQuery, etc.

All of these types of resources are virtually created over the existing data center hardware. These virtual resources can be dynamically allocated to meet varying demands, which is crucial for handling peak loads and scaling down during periods of low activity [30, 31]. This elasticity (dynamic ability to scale and reroute any request) is achieved

through techniques such as load balancing, which distribute workloads across multiple servers and data centers to ensure availability [32, 33].

1.1.2 Cloud Application

A cloud (or cloud-native as they are designed particularly for cloud) application is a software program that a user can access and use directly from their web browser or a mobile app without needing to install it on their computer or smartphone. The application runs on servers that are located in data centers far away from the user, but they can still use it as if it were running on their own device.

Cloud applications are typically composed of small, loosely coupled components known as services [34]. It breaks down the application into smaller, independent parts (services), each responsible for a specific function such as user authentication, data processing, or payment handling. These services create cloud resources, which will be referred to as resources from now on, in my thesis. Services either independently or collaboratively create a resource.

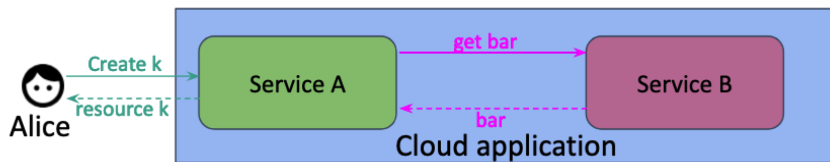

Figure 1.2: Resource creation in cloud application

An example of a cloud application is illustrated in Figure 1.2. It portrays a user *Alice*, requesting *Service A* in a cloud application to create a resource k. To create k, *Service A* requires another resource *bar* from *Service B* which is present in the same cloud application. *Service A* sends a request to *B* for resource *bar* and gets it. *Service A* proceeds to create k with resource *bar* and returns it back to *Alice*. All of this process happens within the same data center.

In a more practical context, *Service B* could be an authentication service and *Service A* could be a payment service in an e-commerce application. To facilitate the payment for a product, the website should authenticate a user to ensure the identity. The payment service will contact the authentication to validate the user and proceeds further. Collaboration of these services together create a single cloud (e-commerce) application.

1.1.3 Service Oriented Architecture

Each service in a cloud application is a self-contained unit that performs a specific function, like processing a payment or authentication in e-commerce. These services communicate with each other over a network, typically using standardized protocols, to create a complete cloud application. This design approach where software applications are built by combining independent, modular services is called Service-Oriented Architecture (SOA) [35, 36].

Since each service is independent, a user can update, replace, or scale them individually without disrupting the entire application. This makes it easier to adapt changing business needs, integrate with other systems, and reuse services across different applications. Services communicate with each other over a network using standard protocols, such as HTTP (Hyper Text Transfer protocol)and REST (Representational State Transfer) [37, 38].

REST API provide a method over the internet for services to interact. REST APIs use HTTP methods such as GET, POST, PUT, and DELETE to perform operations on resources, which are identified by Uniform Resource Locators (URLs). This approach allows for communication, where each request from a client to a server contains all the information needed to understand and process the request. This approach is widely adopted not only by cloud application, but even on the Edge.

1.2 Edge Computing

Edge computing brings the power of cloud closer to the user. Content delivery network (CDN) which cached web contents such as video, audio, etc., is one of the popular example for an edge device [39]. To understand edge computing, it helps to first think about cloud computing, where all the heavy processing and storage of data happen far away in big data center ("the cloud"). Instead of sending the data all the way to these far-off data centers, the processing in edge happens much closer to the user, it could be in the same neighborhood or even within their own device [3, 40, 41].

By processing data closer to where it is generated (like in a smart camera), it reduces the time taken to process data and respond to events. This is crucial for applications that need real-time responses, like self-driving cars or gaming. Edge can be called as Fog computing or cloud-edge continuum (in some cases), it acts as a bridge between the cloud and the small (IOT) devices as illustrated in Figure 1.3 such as Cloudlets [4] or other

solutions such as [6, 42, 43, 44]. It involves deploying a network of intermediate servers that is between small devices and cloud.

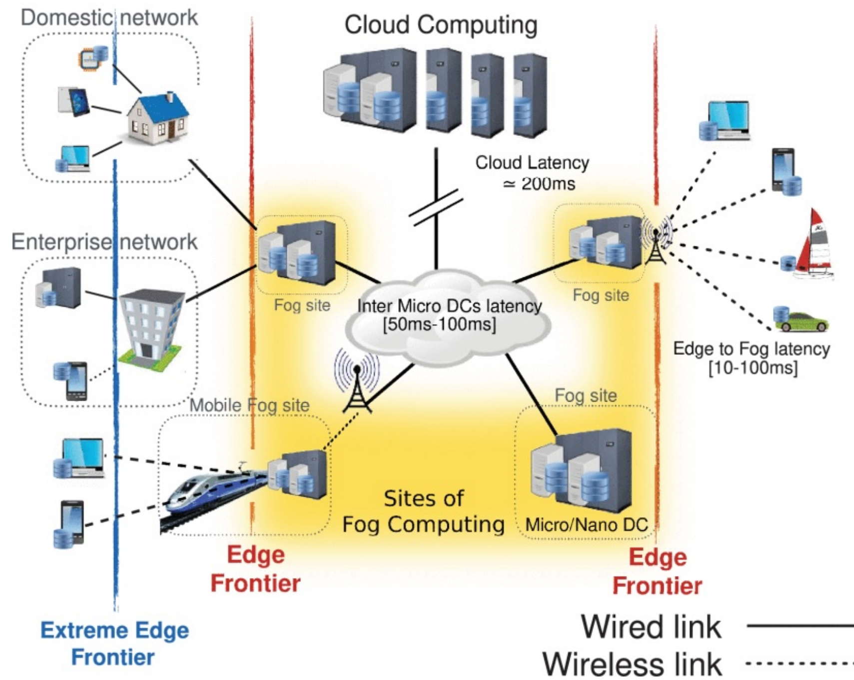

Figure 1.3: Cloud, Edge and Fog computing overview, cited from [45]

These servers can handle data processing tasks that are too large for small devices, but do not require the full power of a cloud data center. It will have better network communication than edge, but less superior than cloud. These are the devices we call edge and target of our solution.

1.2.1 Challenges with Edge

Edge computing introduces several challenges, especially related to networks, primarily because it shifts data processing and decision-making closer to the data source, often at the edge of the network.

Network reliability: In edge computing, data processing is distributed across various devices or sites that are geographically dispersed and connected via a network. These sites often operate in an environment where network connectivity can be inconsistent, such as in remote locations or urban areas with high interference. For example, consider a CDN that distributes video streaming content. In a CDN, edge servers are deployed close to the end-users to deliver the content with low latency. However, these edge servers may

be located in areas where network connectivity is unstable, such as remote regions or areas with poor infrastructure. If the network connection to these devices fail or becomes unreliable, users in that region might experience buffering, delay, or a complete service outage.

These disruptions can be problematic in applications requiring real-time processing of data such as autonomous vehicles or monitoring systems. Maintaining reliable and stable network connections is crucial for ensuring that data is processed and transmitted efficiently between edge and cloud or between different edge devices.

Network bandwidth and latency: Edge computing reduces the need to send all data to a central cloud, but it might require some data to be transmitted over the network to other devices or users, especially for tasks like aggregating results or updating resources. The limited bandwidth at the edge can become a bottleneck, especially when dealing with large volumes of data from devices like high-definition cameras or sensors in smart cities. For instance, in a CDN that delivers high-definition video content, edge servers must manage large amounts of data to stream videos smoothly to users. If the network bandwidth is limited, these edge servers might struggle to keep up with the demand, leading to slower content delivery and reduced video quality. Additionally, if there is high latency in the network, users might experience delays in video playback, leading to a poor viewing experience.

Managing latency is crucial, as many edge computing applications, such as augmented reality or real-time analytics, require immediate responses. Even slight delays in data transmission can affect the performance and user experience of these applications.

1.3 Shift from Cloud to Edge Computing

As discussed, in traditional cloud computing, an application is typically processed in large, centralized data centers located far from the user. While this model offers immense scalability and centralized control, it also introduces significant network latency and disconnections due to their far locations. These issues become particularly pronounced in scenarios where data needs to be processed and acted upon in real-time. For instance, in applications such as autonomous driving, even a few milliseconds of delay can be critical.

Edge computing addresses these challenges by decentralizing data processing, bringing it closer to the data source (often within the same geographic region as the end-user or even on the user's own device). This proximity reduces latency, allowing for faster decision-

making and improved user experience as seen with the CDN example.

However, the shift to edge computing also introduces new challenges. Unlike cloud environments, where resources are virtually unlimited and managed centrally, edge environments are more resource constrained and require more complex management. Network reliability becomes a critical issue, as edge devices often operate in less controlled environment where connectivity can be inconsistent. Moreover, managing bandwidth and latency becomes more challenging as data needs to be processed and transmitted across a more geo-distributed network of edge devices.

Despite these challenges, the move to edge computing is inevitable as the demand for real-time, responsive applications continues to grow. The ability to process data closer to where it is generated not only enhances performance but also opens up new possibilities for applications that were previously limited by the constraints of centralized cloud computing such as autonomous driving. As this thesis will explore, the management of resources in such a geo-distributed environment requires innovative approaches to ensure efficient management, reliability, and scalability.

Chapter 2

From distributed to geo-distributed application

This chapter focuses on presenting the transition from distributed to geo-distributed systems, which is directly linked to the shift from cloud to edge. I discuss the existing solutions to geo-distribute an application. This section presents an initial categorization of these solutions and addresses their limitations. These limitations and the identified problems led us to the initial research work in our STACK research group, published at the Euro-Par 2021 conference [11].

2.1 Distributed Cloud Application

In a cloud application, resources such as data storage and computation (as explained in Section 1.1.1) are spread across multiple nodes or servers within a single DC with SOA (explained in Section 1.1.3). They are managed by loosely coupled components called services, as discussed in Section 1.1.2. These services work together to appear as a single cohesive unit (cloud application) to the end user, masking the complexity of the underlying infrastructure. The primary goal of such a distributed architecture is to ensure that the cloud application can handle varying loads (scalability), maintain availability, and recover quickly from failures.

To achieve high availability, these applications are designed with redundancy at multiple levels. This means that even if a particular node or server fails, the application continues to function without interruption by rerouting tasks to other available nodes. For instance, in a distributed cloud storage system, data is often replicated across multiple servers within a data center. If one server goes offline due to hardware failure, the system can automatically switch to a replica on another server, ensuring that users can still access their data without experiencing any downtime. This redundancy is crucial for maintaining the reliability of cloud resources, especially for applications that require

constant uptime, such as e-commerce platforms or online banking systems.

Another significant advantage of the distributed nature of cloud applications is scalability. Cloud applications can dynamically scale their resources up or down based on demand. This elasticity is made possible by the distributed architecture within the data center, where additional computing resources can be allocated to handle increased loads. For example, during peak times, an e-commerce application can automatically provision more servers to manage the surge in user traffic, ensuring a smooth user experience.

However, as the demand for global reach and low-latency resources has grown, the limitations of traditional distributed systems have become more apparent, arising the question: How can we extend an application outside of one region? It could be for more availability of the application across multiple regions, or to make it more reliable (if a data center crashes), or to process data closer to the user, such as in edge.

2.2 From distributed to geo-distributed applications

Geo-distributed cloud applications extend the principles of distributed computing across multiple geographic locations, often spanning continents. In this model, data and computation are not just distributed within a single region but are spread across several data centers worldwide. This approach is driven by the need to bring computing power and data storage closer to end-users, reducing latency, improving the reliability and availability of applications for users in different regions.

For example, a global social media platform (such as Facebook) may store user data in data centers located in different countries to ensure that users experience fast load times and minimal delays, regardless of their location. Geo-distribution also enhances resilience by ensuring that the failure of a data center in one region does not disrupt service globally, as other data centers can take over seamlessly.

We analyze many solutions to geo-distribute an application. The detailed analysis is presented in Chapter 4. In this section, I talk about a few relevant works that can help us categorize them. We categorize them into three (as illustrated in Figure 2.1):

- *Inter-Service collaboration:* Geo-distribution of an application occurs through the collaboration of services across different instances.

- *Broker based collaboration:* Geo-distribution of an application occurs through the collaboration of dedicated brokers acting as middleware between services across

different instances.

- *Database Collaboration:* Geo-distribution of an application occurs through the collaboration of geo-distributed databases across different instances.

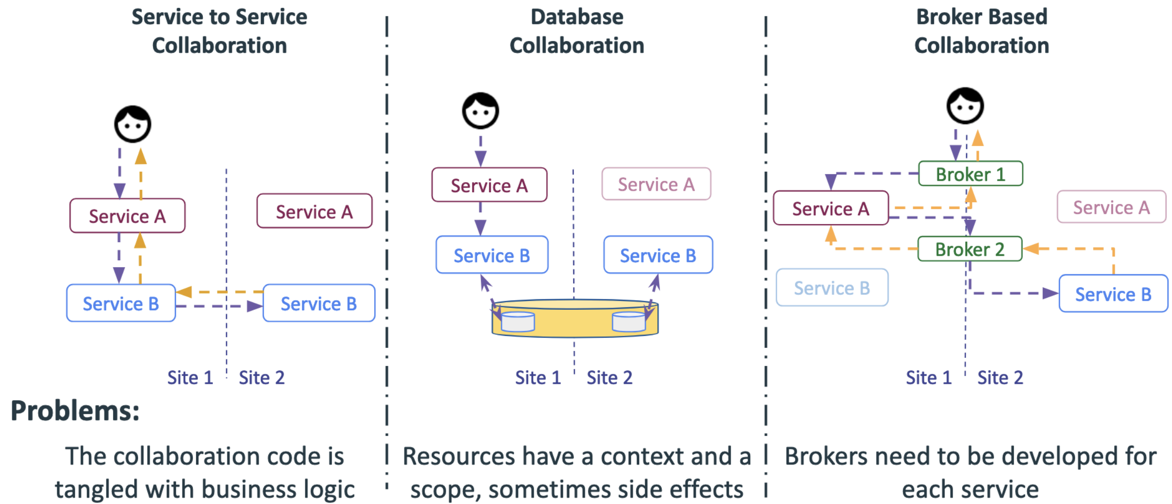

Figure 2.1: Existing approaches for Collaboration and their issues

I further explain these categories with relevant existing approaches:

2.2.1 Inter-Service collaboration

Scenario: Geo-distributing OpenStack with Federated Keystone.

OpenStack is typically deployed in a single data center or location, which limits its ability to serve users or workloads spread across multiple geographic regions. This limitation can be challenging as a user can have latency or downtime to access a software deployed on one OpenStack instance. To address these needs, efforts like Federated Keystone [8, 46] have been developed to enable geo-distribution of softwares deployed on OpenStack.

Keystone is an identity management service within OpenStack for any resource deployed on it. Federated Keystone is an extension of this service that allow multiple OpenStack environments located in different regions to work together as a unified application.

For example, consider the same global e-commerce platform deployed on OpenStack. Users located in France and India might want to ensure that they can authenticate once and access resources in both locations without needing to log in separately to each data

center locations. Federated Keystone enables this by linking the identity services of the different OpenStack instances, allowing users to move between regions.

Setting up Keystone federation involves configuring multiple OpenStack clouds to recognize and trust each others identity services. Beyond the initial configuration, ongoing management and maintenance are also necessary to ensure that the federated Keystone setup remains secure and functional as the environment evolves. An overall modification from the original code of the service keystone is required to facilitate this geo-distribution.

2.2.2 Broker based collaboration

Scenario: Geo-distributing Kubernetes application with Kubefed.

Kubernetes is a powerful application for running and managing containers, but by default, it typically operates within a single location or region. This setup can be limiting for deployed resources that need to be available across different parts of the world. Geo-distributing an application can help to solve these challenges, however, Kubernetes alone doesn't support this out of the box. This is where KubeFed (Kubernetes Federation) [47] comes in, enabling management of multiple Kubernetes clusters spread across different locations as if they were a single application.

Consider a global e-commerce platform. If this platform is running on a Kubernetes cluster in France, users in India might experience latency while loading the platform. With KubeFed, the platform can be replicated to regional clusters, located in each of these countries, ensuring that all users will have fast and reliable access.

KubeFed can be thought of as a broker that facilitates the seamless coordination and management of multiple Kubernetes clusters across different geographic locations. Just like a broker who manages and negotiates resources between various parties, KubeFed oversees the distribution of workloads, resources, and configurations across several clusters, ensuring they work together as a unified application.

KubeFed require adjustments in how resources are defined and managed within Kubernetes. For instance, certain Kubernetes objects, such as deployment, service and namespace, requires additional configuration from the user to initiate geo-distribution. This might involve modifying existing Kubernetes configurations or using KubeFed-specific annotations. These configuration changes are essential for KubeFed to function and maintain the desired resource at each site.

2.2.3 Database collaboration

Scenario: Geo-distributing Kubernetes application with a database.

As discussed in Section 2.2.2, Kubernetes application by default operate in a site or a data center. We already saw a broker based approach with KubeFed to geo-distribute a Kubernetes application. Similarly, another approach to achieve the same is with a geo-distributed database [48]. By integrating Kubernetes with a geo-distributed database, each cluster can maintain autonomy while still participating in a larger, cohesive system.

In this solution, a geo-distributed database Riak [49] is used. It is built on conflict-free replicated data types (CRDTs) [50], enabling consistent and reliable data sharing across clusters without the need for constant communication with a central controller. The configuration of each resource in Kubernetes like a pod, secret, deployment, etc., are stored in the database and these values are geo-distributed among different instances of an application. If any change in the DB value for a resource is detected, a custom controller at a site access this data and modify its Kubernetes cluster and later synchronize the values with replicated DBs.

One significant challenge with using a geo-distributed database in a application like Kubernetes is managing configuration values that are specific to a particular site or cluster. These values should not be propagated across other instances. For example, in a pod configuration, there will be annotations representing the name of the worker node it operates. This name is local to a Kubernetes cluster and it could be different at each instance.

Another example, consider a Kubernetes resource ConfigMap, it is used to store configuration data for a resource deployed in a cluster. In a geo-distributed Kubernetes setup, this ConfigMap might contain environment-specific values, such as API endpoints, database connection strings, or region-specific service URLs. If this ConfigMap were replicated across all sites using a geo-distributed database, it can lead to unintentionally use of configuration values from remote instances that could lead to misconfigurations or service disruptions.

An analysis for the same approach with an OpenStack VM and Image was done by the STACK research group and published in the article [11].

2.2.4 Limitations with existing approaches

These categories propose different approach towards geo-distribuing an application. As illustrated in Figure 2.1, they come with their own set of limitations. As mentioned earlier, collaboration between instances is necessary for geo-distribution. We identify them for each of the collaboration:

- *Inter-Service collaboration:* Most of these services such as Keystone in OpenStack are not designed for geo-distribution between multiple instances. To implement such a method, it requires dedicated code written over the existing business logic of the keystone service.

- *Broker based collaboration:* The brokers are generally designed for an application or specific for a service, such as KubeFed for Kubernetes. They are not generic to any application.

- *Database Collaboration:* As highlighted, there could be values that are local to a site, if these are geo-distributed, it can create disruptions at a remote site.

Given the limitations, there is a clear need for a more generic and non-intrusive solution to geo-distribute applications as studied in detail by our team and published in [11, 13, 14, 17]. Such a solution should seamlessly integrate with any application without requiring extensive modifications to the codebase. I define the main research question, that is focused by our STACK team and laid the foundations for my research:

Research Question : Is it possible to create a solution to geo-distribute any application without being intrusive to the existing code base?

A generic and non-intrusive geo-distribution solution would significantly reduce the development and maintenance burden on developers. It would enable faster adoption of geo-distribution capabilities across a wide range of applications, from legacy till modern cloud-native systems. Additionally, by decoupling geo-distribution logic from the application code, our approach would enhance the flexibility and scalability of the application, making it easier to adapt to changing requirements and technological advancements, especially in edge systems.

Inherent advantage of broker based approach

We categorized the existing approaches to three, as portrayed in Figure 2.1. To create a non-intrusive and generic geo-distribution approach, broker based approach particularly stands out among these category. A broker is often placed external to a service, opening a potential for them to be able to geo-distribute an application externally. In our survey, solutions that utilize Broker based collaboration, such as Kubefed [47], KubeEdge [9], OneEdge [51], etc., are designed to be intrusive and specific to an application.

The reason why the existing broker based approaches are non-generic is because they are currently designed with an application aware context, i.e., the broker contains specific APIs and knowledge of application itself. For example, Kubefed knows about Kubernetes configurations and compiles the configuration external to the application with the knowledge it posses. We can create a broker that does not contain the application knowledge, making it generic to any.

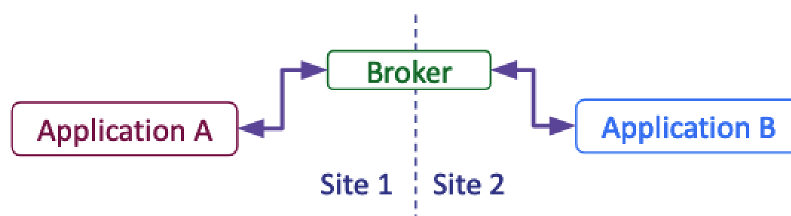

Figure 2.2: A basic Broker based design

A broker based design as portrayed in Figure 2.2, consist of an instance communicating with a broker to facilitate geo-distribution of an application. It contains an external piece of software that has a certain logic to perform geo-distribution. A broker externalizes any function it is given, such as geo-distribution.

As discussed in Section 2.2.4, an inter-service collaboration is intrusive and a database collaboration has the potential to create disruptions. In our initial survey, a broker based approach has the potential to create a generic and non-intrusive solution for geo-distributing an application.

2.3 Summary

This chapter provides a comprehensive exploration of the transition from distributed to geo-distributed applications, highlighting the evolving needs and challenges associated

with extending applications across multiple geographic regions. The discussion began by examining the nature of distributed applications within a single data center, where redundancy, scalability, and fault tolerance are achieved through the strategic distribution of resources across multiple nodes.

As the chapter progressed, we explored the concept of geo-distributed applications, which take these principles a step further by spreading resources across multiple data centers worldwide. This approach enhances application availability and reliability for a global user base, reducing latency and ensuring continuity even in the event of regional failures.

The chapter also provided an initial categorization of existing approaches to geo-distribute an application, including inter-service collaboration, broker based collaboration, and database collaboration. Each approach was analyzed in terms of its benefits and limitations, laying the groundwork for the development of a more generic and non-intrusive solution for geo-distributing applications. The inherent advantages of broker based collaboration were particularly emphasized, offering a promising direction for our research.

Chapter 3

A solution to externalize geo-distribution

This chapter focuses on presenting an initial approach by the STACK research group to address the limitations identified in the previous chapter, i.e., to geo-distribute an application in a generic and non-intrusive way. The concepts discussed in this chapter are exerts from the PhD dissertation [12] of my colleague Marie Delavergne, who defended her thesis on March 2022. I will explain the major components presented in our initial approach and proceed to identify the limitations in them. Later, I highlight the research questions from these limitations that led to my PhD thesis.

3.1 Creating a generic and non-intrusive solution

As highlighted in Chapter 2, the existing solutions are either intrusive or not generic enough for any application. A cloud application, as described in Chapter 2 and Chapter 1, consists of a modular design that combines multiple services to manage a resource. Each service is designed to perform a specific function and can communicate with other services via REST APIs as described with the SOA in Chapter 1.

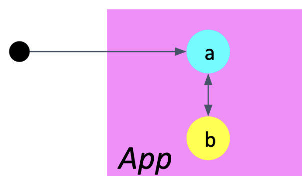

Figure 3.1: A cloud application *App* with two services *a* and *b* along with a user (represented with a black dot) initiating the request

Figure 3.1 portrays a cloud application with a request from the user to create a resource

from service a, which requires another resource from service b. Service a need to contact b to get the resource to complete the user request. The workflow for this request is: 1) $user \rightarrow a$ 2) $a \rightarrow b$ and for return: 1) $b \rightarrow a$ 2) $a \rightarrow user$. As mentioned in Section 1.1.1, resources in the context of a cloud application can be anything that the application manages, such as data objects, compute instances, storage, network configurations, etc., and they can be managed via APIs (like REST) developed for them.

A first approach proposed by the team towards geo-distributing an application is to deploy a complete instance at each site. An application instance deployed at each of the involved site helps to satisfy any local requests (ensuring autonomy) and the ability to perform under network partition between them. However, having only autonomous instances at each site implies that these instances cannot share resources with each other.

An instance of application may not have enough resources to complete all the requests, as mentioned in our early research [11]. Consider an OpenStack application that is geo-distributed across *Site* 1 and *Site* 2. It follows the Cheops geo-distribution approach, in which a complete instance of the application is deployed at each site. Each site can handle user requests locally, unless a request requires a resource that is not available at the local site. In such cases, coordination with other sites is necessary to fulfill the request.

For example, creating a VM in an OpenStack application at *Site* 1, requires an image that is not available locally. The image is only available at *Site* 2, portraying that sharing is required between these sites to complete the request. Having only autonomous instance cannot help to satisfy this request.

Hence, to satisfy these requests, collaboration approach was introduced into our solution to facilitate sharing of resources between independent instances of an application. Figure 3.2 illustrates a collaboration for a request to service a initiated at *Site* 1 that requires a from b at *Site* 2, same as the VM-Image case.

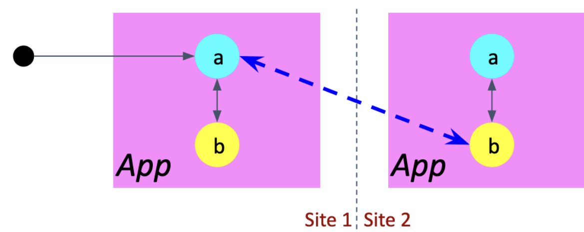

Figure 3.2: Two instances *App* with two services a and b deployed on two sites, where service a from *site* 1 depends on b from *site* 2

Here, I present the first two principles for our proposal:

- ***Local-first***: A complete instance of the application is deployed at each participating site, enabling local decision-making and autonomous operation without relying on other sites.

- ***collaborative-then***: In addition to having a complete instance, each application instance has the capability to collaborate with other instances when needed to fulfill a request.

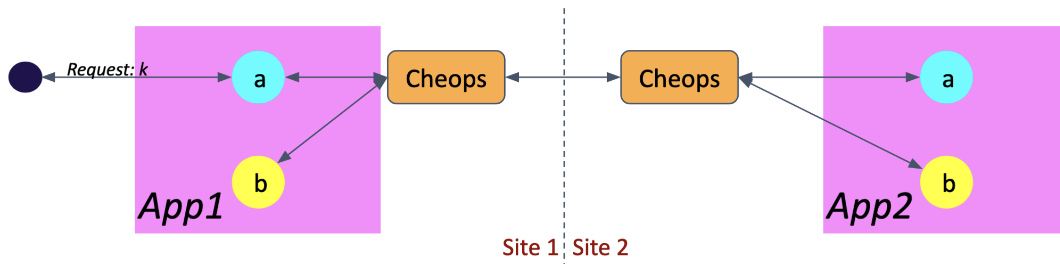

Figure 3.3: Two instances of a cloud application *App*; user makes a request k on Site 1, it is forwarded to the remote instance with Cheops

Figure 3.3 portrays our proposal, entitled cheops. It acts as a middleware between two independent instances of an application. As discussed in Section 2.2.4, the inherent design advantage of a broker being external to an application lead us to creating this middleware.

A copy of our middleware is deployed at each of the involved application instances. Each of the application instance are independent and Cheops creates the illusion of a single application across these instances. All the request from the user is sent to Cheops and it initiates collaboration between the instances, if required, based on the request. Within these instances, it can route the request to any of the required local services. For example, a request k from *Site* 1 in Figure 3.3 can redirect it to service b at *Site* 2.

Cheops is designed to be non-intrusive and generic to any application as mentioned in Section 2.2.4. Adopting a widely used protocol such as REST API and relying only on application API's for communication makes it generic to any application. Cheops relies on the broker design approach, i.e., it is external from the code of the application business logic. It runs as an independent software outside the application.

Each of the application API's are different, for example, Openstack will have a set of API's different from Kubernetes. Since Cheops is generic, it only forwards the application request as required by the user, but how to define these requirements? To facilitate geo-distribution, the user needs to provide some additional details along with each request.

Our team has designed a simple and generic Domain Specific Language (DSL) called scope-lang [11] that can be attached to any existing application REST API. This creates a common interface for the user to define the geo-distribution requirements irrespective of the application. Scope-lang consist of collaboration and site details.

For example, a normal openstack application to create a vm *bar* is `openstack server create --image centos foo` , this is can be applied directly to a local instance. If the image centos is not available locally, Cheops can facilitate a request to get the image from a remote site (*Site* 2) with the scope-lang. The OpenStack request with the scope-lang expression from at *Site* 1 will be: `openstack server create --image centos foo -scope {compute: Site 1, image: Site 2}`. This is sent to Cheops directly and it interprets the scope-lang expression to understand the geo-distribution requirements and separates the application request.

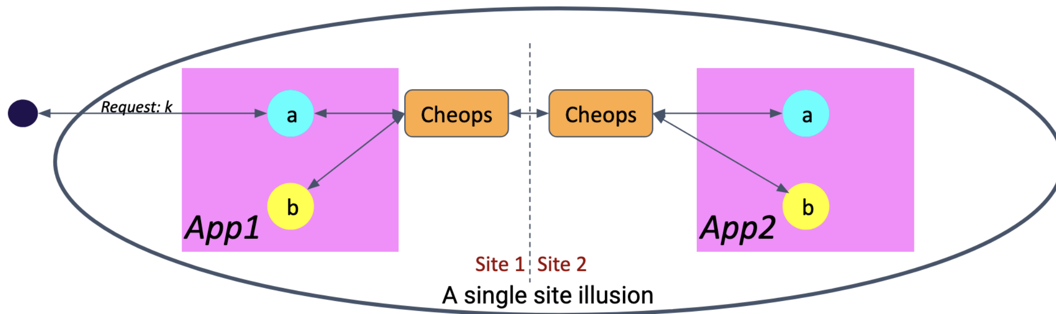

Figure 3.4: A single site (unified) abstract view created from two instances of a cloud application with Cheops

The interconnectedness and collaboration between instances creates a unified and coherent system, as illustrated in Figure 3.4. This concept of a single coherent system, as described in the definition of distributed system, is analogous to the Single System Image (SSI) research [16]. It contained individual physical systems that are connected together via network to create an illusion of a single system. In reality, the system is composed of a collection of distributed and heterogeneous operating systems connected together.

Cheops has a similar concept, where it tries to create a single system by connecting individual and independent instances of application. All of the involved instances will present the same result for any query. For example, a request sent by a user to Cheops from either *site* 1 or *site* 2 will get the same response because of the unified single system. The communication and routing between these instances to fetch resources, are completely managed by Cheops, as per the scope-lang expression.

In 2022, my colleague Marie Delavergne defended her thesis [12] on Cheops. She focused on the initial design including request-forwarding, scope-lang, single system abstraction and an initial set of collaborations. I will describe in short about the collaborations in Cheops, followed by the scope-lang and later, the limitations with the existing approach.

3.1.1 Collaborations

Collaborations in Cheops consist of various approaches to geo-distribute a resource across multiple independent instances of an application. At the moment, there are two well defined and one pre mature collaboration in Cheops. The first two collaborations called Sharing and Replication are defined and published in the articles [11, 12, 13, 14].

Each of these collaborations in Cheops is expressed using **Scope-lang**, a Domain Specific Language (DSL) developed specifically for Cheops to facilitate geo-distribution. A Scope-lang expression includes details about the sites and collaboration requirements necessary to geo-distribute an application. This expression is sent along with the application API, meaning it operates external to the application and ensures that no modifications to the business logic code are needed. I will explain the Scope-lang expression for Sharing and replication collaborations, along with their explanation below:

Sharing

Sharing defines a dynamic composition of services between instances of a geo-distributed application. It presents the ability to forward a request to remote instances or any specific endpoint of the service in the application. This approach is highly motivated by the fact that a resource may not be available on every site, as discussed early in Section 3.1, and sometimes it requires a sharing between instances to complete a request.

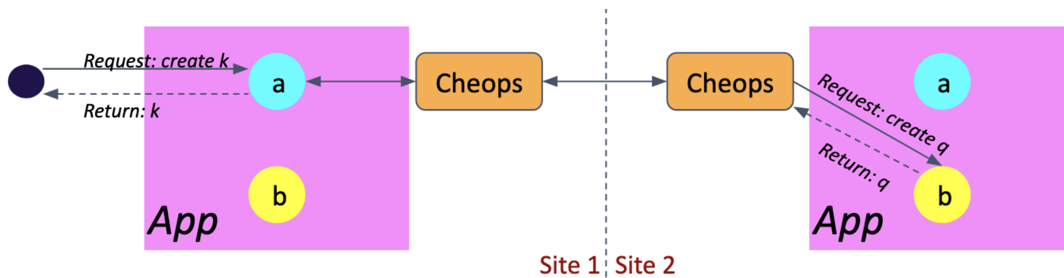

Figure 3.5: Two instances of a cloud application *App*; user makes a request k on site 1, it is forwarded to the remote instance with Cheops

An example is illustrated in Figure 3.5, a request is send by a user to create a resource k from *Site* 1. The scenario is to create k, the *App* requires another resource called q from service b, which is not available at *Site* 1.

An example for *Sharing* portrayed in Figure 3.5 and the workflow for it will be:

1. A user sends a request to Cheops to create resource k from *Site* 1 by fetching another resource q from *Site* 2 service b. The scope-lang expression for this request will be `application create k --sub-resource k -scope {a: Site 1, b: Site 2}`.

2. Cheops receives the request and parses the scope-lang expression.

3. From the parsed expression, Cheops determines that it is for replication collaboration (due to the & symbol used), resource k needs to be created by service a at *Site* 1, and that resource q must be fetched from service b at *Site* 2.

4. It initiates the creation of resource k by sending the request to service a at *Site* 1.

5. Simultaneously, it forwards a request to the instance at *Site* 2 to retrieve resource q from service b.

6. Once resource q is obtained, it is sent back to service a at *Site* 1.

7. Finally, Cheops completes the process by returning resource k to the user created by service a at *Site* 1.

Replication

Replication creates and manage identical copies of a resource. The existing solutions to replicate a geo-distributed resources are either specific to an application or requires additional code added on its business logic (as discussed in Section 2.2.4). Cheops *Replication* presents an approach that can perform a geo-distributed replication at the API level. This makes it non-intrusive and generic to any REST API based application. The approach was validated in the thesis of my colleague, Marie Delavergne.

The workflow for replication with an example portrayed in Figure 3.6, will be:

1. A request to create a resource t is initiated at *Site* 1 instance of *App* by the user. The request syntax is `application create t -scope {Site 1 & Site 2}`.

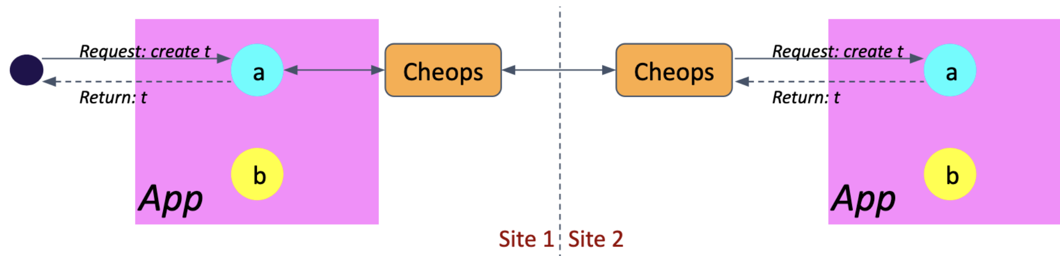

Figure 3.6: Two instances of a cloud application *App*; user makes a request k on site 1, it is forwarded to the remote instance with Cheops

2. The request is received by Cheops and it parses the scope-lang expression to identify that it is for replication collaboration (due to the & symbol used) and the request needs to be replicated at *Site* 1 and *Site* 2.

3. Cheops forwards the request to the subsequent agents (including a local site in this case) at each involved site.

4. Cheops collects the responses from these agents and returns it to the user.

The replication collaboration relies on a RAFT-like protocol to ensure consistency. RAFT is a consensus protocol that only applies an operation if a minimum number of replicated instances (quorum) agree on it. While this approach offers a consistency model that is both generic and external to the application, it falls short in scenarios involving network partitions, where achieving quorum becomes impossible.

To address this challenge, she introduced a rollback mechanism for each operation. When an operation is initiated at the local site, it is applied immediately before attempting to obtain a basic quorum. Once quorum is sought, one of two outcomes occurs: if successful, no further action is needed; if quorum is not achieved, the resource is rolled back to its previous state before the operation. While this method is effective, it can be costly to create a roll-back method for each operation and it requires knowledge of the application.

3.1.2 An updated Cheops principles

The core of our solution lies in four major principles:

- **Non-Intrusive**: Geo-distributing an application should not require adding new code to its existing business logic.

- **Generic**: Geo-distributing an application should be applicable to any.

- **Local-first**: Each instance in a geo-distributed application should have the autonomy to execute any operation locally without relying on other sites.
- **Collaborative-then**: While autonomous, these application instances should be able to collaborate with each other when needed to share resources and complete operations.

STACK research group built the Cheops framework with these design principles.

While these principles provide a strong foundation for building a geo-distribution solution, there are still significant challenges that need to be addressed in practice. In the following section, we will explore the limitations of the current Cheops approach, highlighting areas that require further research. These challenges form the basis of my thesis and the work presented ahead.

3.2 Limitations with our existing approach

Here, I will talk about the limitations with our current Cheops approach and why we need to do further research to solve them. There are three major research problems we identified:

3.2.1 Issues with replication in general

Traditional approaches to geo-distribute a resource predominantly rely on replication, where full copies of resources are maintained across multiple locations. For example, Content delivery networks (CDN) [39], as discussed in Chapter 1, replicate web content, such as images, videos, and static files, across multiple servers worldwide. When a user in France access a website, they receive content from a server located in France, rather than one from India, reducing the latency. Similarly most of the databases such as Google Spanner [52], object stores such as Amazon S3 buckets [53] or databases like Couchbase [54] follow the same.

Replication, though robust and widely adopted, can lead to several issues. The issues highlighted are for replication in general, not for Cheops based replication in particular. The issues are:

- *High Synchronization Overhead:* Synchronizing a resource across multiple locations requires frequent communication with all involved instances to ensure consistency.

For example, in a geo-replicated Kubernetes application, any change to a resource, such as a pod configuration, must be propagated and synchronized across all sites. This constant synchronization significantly increases bandwidth consumption, particularly as the number of sites grow. For edge systems like CDNs, which often operate with limited bandwidth, this can result in severe network congestion and bottlenecks [55].

- *Increased latency:* Replication tends to increase latency, particularly in geo-distributed environments. Since each site must maintain an identical copy of the resources, any request sent to one instance must traverse large distances to ensure consistency [56]. This long-distance transfer can introduce delays, affecting the responsiveness of the application. For latency-sensitive applications, such as real-time analytics or online gaming, these delays can significantly degrade the user experience.

- *Risk of system-wide failures:* Replication can also make the system more vulnerable to wide-scale failures. In a fully replicated system, a failure or inconsistency in one instance can potentially propagate to others as they attempt to synchronize with the problematic instance. This interconnectedness means that an issue in one part of the system could trigger a cascading failure, affecting the entire geo-distributed environment. Such failures can be challenging to resolve, as they may require significant effort to identify and correct inconsistencies across all instances.

- *Full Copy Replication:* Replication is often resource-intensive, as each instance is required to store a complete copy of the entire resource, regardless of its relevance to a particular region. This duplication leads to increased usage of storage and computational resources, especially at the edge, where resources are often more limited. While full replication may be necessary in certain scenarios, there are many cases where only a portion or subset of a resource is needed at a specific edge site, based on user proximity. For example, users in a country are more likely to consume content relevant to their region, which is why services like Netflix configure their CDNs to prioritize regional content. A CDN in France and one in India will host different content to suit their respective audiences. A full replication approach, however, does not account for such regional variations, as it assumes identical resources across all sites, leading to unnecessary replication and resource consumption.

3.2.2 RAFT based consensus may not be enough

Current version of Cheops implement a RAFT like consensus protocol for consistency as discussed in Section 2.2.4. It relies on a rollback mechanism to ensure a local-first nature along with the RAFT approach. While RAFT is effective in maintaining consistency within a distributed system, it presents significant challenges when applied to geo-distributing resources across multiple, geographically dispersed sites.

An operation is locally applied first due the local-first nature of Cheops. As discussed in Section 2.2.4, if this operation fails to acquire majority during the voting phase in RAFT, the resource is rolled back to a previous state before the operation is applied. A combination of RAFT and rollback can ensure a strong eventual consistency, but it has its issues:

- *Risk of Partitioning and Failures:* a significant concern is the risk of network partitioning, which is more likely in a geo-distributed system due to the varied and potentially unreliable network conditions across different regions. RAFT relies on continuous communication between nodes to maintain consensus. If a network partition occurs, separating sites from one another, RAFT may fail to achieve consensus, leading to the system becoming unavailable or inconsistent for a moment. This risk is particularly problematic in critical applications where downtime or inconsistency could have severe consequences.

- *Issue with Rollback:* Implementing a rollback mechanism can introduce additional complexity and overhead, especially when dealing with geo-distributed systems. In the event of a failure or inconsistency, rolling back an operation requires reverting resources to a previous state. However, in geo-distributed environments, ensuring that all instances involved in the operation can correctly and efficiently rollback adds significant challenges. This is particularly problematic when the rollback process depends on the application business logic, as it may require custom implementations for each specific operation and resource. Additionally, frequent rollbacks due to network partitions or other issues can lead to inconsistent states across different sites and a degradation in system performance, making the approach less reliable in practice.

- *Latency and Performance Issues:* One of the main challenges with using RAFT in a geo-distributed environment is the increased latency it introduces. RAFT requires

that all updates be agreed upon by a majority of nodes, which necessitates communication across all participating sites. In a geo-distributed system, these sites may be spread across vast distances, potentially on different continents. The time it takes for messages to travel between these sites can be substantial, leading to delays in reaching consensus. This latency can severely impact the performance of applications that require rapid updates and low response times, making RAFT-based consensus less suitable for such environments.

3.2.3 Difficulty in creating an illusion of a single application

As mentioned earlier in Section 3.1, Cheops creates an illusion of a single application by combining individual geo-distributed instances. This implies that all of the operations that are possible in a single instance of an application should be possible with Cheops illusion.

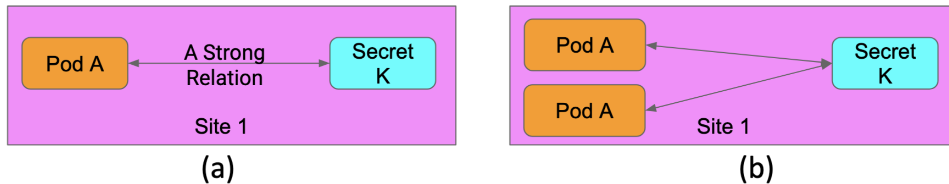

Figure 3.7: Single site cloud application portraying :(a) a strong relation between secret and pod (b) replicating the pod in the same site

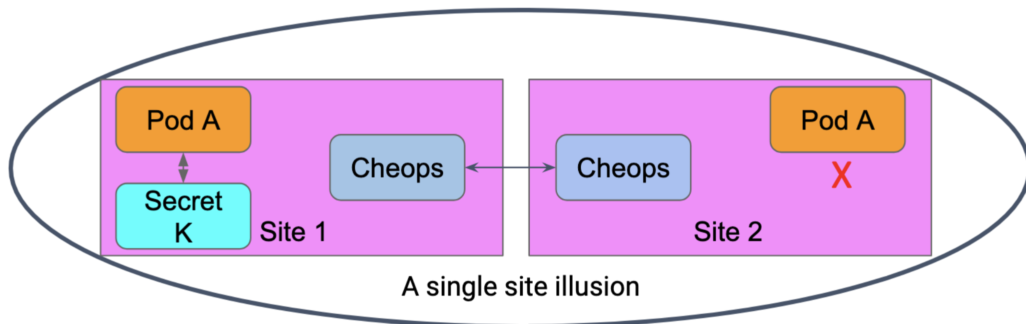

Figure 3.8: The illusion provided by Cheops fails as the pod fails when trying to geo-distribute it to Site 2, due to the absence of dependent secret resource

For example, in a single instance of Kubernetes, a pod can have a strong relation with another resource called secret, as illustrated in Figure 3.7 (a). A strong relation indicates

that if the secret is not available, the pod can lead to a failure state. If we try to replicate this pod within the same instance, it will be successful, as it can attach to the existing secret, as illustrated in Figure 3.7 (b). The process of replication works within an instance.

If we try to perform the same replication across different instances with Cheops, i.e., for example, if a user tries to replicate a pod attached to a secret from *Site* 1 to *Site* 2, as illustrated in Figure 3.8. The process of replication will fail at *Site* 2, as there is no secret available there. This example (Figure 3.7 (b) and Figure 3.8) contradicts the illusion of having a single application, as the results are different for a single instance and Cheops illusion. This is a strong problem to the envisioned Cheops approach to create an illusion of a single application.

3.3 Research Questions

In my thesis, I raise three different research questions:

Research Question 1: Is it possible to design an alternative collaboration method that addresses replication challenges while adhering to the Cheops principles of prioritizing local-first, non-intrusive, and application-agnostic principles?

Research Question 2: Can we conceptualize an approach for consistency that overcomes the limitations of the existing Cheops approach, while aligning with its principles of being local-first, non-intrusive, and applicable to any application?

Research Question 3: Can we guarantee that operations within a single instance can be successfully reproduced in a Cheops geo-distributed environment, thereby creating the seamless illusion of a single application?

3.4 Summary

In this chapter, we explored the initial solution developed by the STACK research group to address the challenge of geo-distributing applications while maintaining a generic and non-intrusive approach. The solution, Cheops, is a middleware based approach that ensures

that it is non-intrusive and generic to any applications. Each instance of an application is autonomous with collaborative features.

We examined each component of the Cheops solution, highlighting how it enables sharing and replication between independent instances of an application. Despite its promising approach, Cheops fails to address certain issues, especially for replicating a resource (in general), consistency under network partitions, as well as in fully realizing the illusion of a single cohesive application across geo-distributed sites.

From these limitations, we have identified key research questions that lay the foundation for my thesis. These questions focus on developing a new collaboration method, consistency approach, and ensuring seamless operations for the illusion created across geo-distributed instances. Subsequent chapters will address these research challenges and outline potential solutions to overcome the limitations identified here.

Part II

State of the art

The second part of my PhD thesis delves into the various approaches to identify the most suitable solution for the research questions outlined earlier. I have analyzed these solutions to develop a comprehensive survey, which serves as the state-of-the-art contribution in my work. The survey is divided into two chapters, each addressing one of the research questions.

In Chapter 4, I provide an analysis of middleware based approaches to geo-distribute an application. It is aimed to address the first research question of my thesis, as described in Section 3.3. Multiple solutions were evaluated against our specific requirements to determine their potential applicability in my work. The comparison includes identifying their collaboration mechanisms, particularly focusing on issues like synchronization overhead.

In Chapter 5, I investigate solutions that geo-distribute application with a focus on their consistency approach. It is aimed to address the second research question of my thesis, as described in Section 3.3. Each solution is assessed based on our synchronization needs, with the goal of identifying a potential fit for my research.

Chapter 4

Existing geo-distribution solutions

This section compares different solutions that are designed to geo-distribute an application. The primary goal of this section is to identify a possible solution to answer the first research question, as described in Section 3.3. This answer need to satisfy our principles, as described in Section 3.1.2. I analyze each solution against a set of comparison points that can potentially lead us to find one that can address the limitations posed in this research question.

4.1 Design for our approach: Research Question 1

The core design of Cheops is to offer a middleware that facilitates geo-distribution of an application. One of the main component is the collaboration model it offers. Each collaboration model employs a distinct approach to achieve geo-distribution. In Sharing, resources are forwarded between sites as required, while in Replication, they are duplicated across multiple sites.

In modern applications that utilize Service Oriented Architecture, multiple services need to collaborate to create a geo-graphically distributed environment. A prominent approach that facilitates such a setup is Orchestration [57, 58]. Orchestration automates the deployment, configuration, and maintenance of these services, eliminating the need for manual intervention at every step [59][60]. In Cheops replication, orchestration enables the collaboration to manage these aspects of geo-distribution, while following the principles outlined in Section 3.1.2.

Research Question 1 seeks to explore a new collaboration approach that aligns with our core principles. Replication has already proven that orchestration design can help us create the required collaboration [12]. Hence, to design our new collaboration, we utilize the same design. In this chapter, we evaluate various approaches based on orchestration

and middleware design to identify their collaboration approaches to geo-distribute an application. We check if an existing solution can address the first research question.

4.2 Comparison Points

This section presents a list of comparison points that I used to analyze various solutions. These points are chosen to match the requirements we are looking for in our required collaboration. A consolidated view of all the approaches examined is presented in Table 4.1. To improve clarity and readability, certain comparison points have been abbreviated. Their short notations and detailed explanations are provided alongside each of them.

Coordination type (CT): represents the architecture of the solution. They include:

- Centralized: The solution relies on one instance to make the decisions to geo-distributed an application.
- Decentralized: The solution relies on more than instance (usually one at each site), to make decisions to geo-distribute an application.
- Hybrid: A mixture of both.

Local-first (LF): represents the autonomous nature of a solution.

- Yes: An operation can be initiated at any site and be executed locally even during network partition.
- No: An operation cannot be executed locally without coordination from other sites.

Collaborative-then (Collab-T): represents the collaborative nature of a solution. It facilitates sharing of resources between multiple geo-distributed instances.

- High: The solution allows all of the resources in an application to be collaborative.
- Medium: The solution allows only a set of resources in an application to be collaborative.
- Low: The solution allows one or a few resources in an application to be collaborative.

Generic: represents the generic nature of a solution. It facilitates managing multiple applications with the same solution.

- High: The solution can be used to geo-distribute any application.
- Medium: The solution can be used to geo-distribute a set of applications.
- Low: The solution is specifically designed for an application.

Non-intrusive (NI): represents the non-intrusive nature of a solution.

- High: No code change in business logic of an application is required to geo-distribute an application.
- Medium: Some code changes are required, such as a new configuration to geo-distribute an application.
- Low: Need to change the core application business logic to geo-distribute them.

Partition handling (PH): represents the capability of an application to perform under network partition between instances.

- High: Local application and operations can perform autonomously under network partition.
- Medium: Local application and some operations can perform under network partition.
- Low: Local application can perform in a read-only mode under network partition.

Synchronization overhead (SO): represents the additional coordination in communication required during synchronization between application instances.

- High: Requires coordination between instances of an application to perform an operation.
- Medium: Requires coordination between multiple sites, only in the event of conflicts between operations or after network partition.
- Low: Requires very less (close to none) coordination between instances of an application to perform an operation.

4.3 Existing solutions Collaboration

In this section, I explore existing solutions that could serve as potential candidates for our new collaboration model. The presentation of each solution includes an explanation, followed by a comparative analysis with our envisioned approach, and concludes with an evaluation of whether it can be adopted or not.

4.3.1 HYDRA: Decentralized Location-Aware Orchestration of Containerized [61]

Hydra is a decentralized solution for geo-distributing container-based applications. It was developed as an alternative to Kubernetes, which follows a centralized approach for managing containers. Hydra adopts a fully decentralized approach with locality-aware scheduling, allowing users to specify the location where a resource should be deployed. The solution includes replication collaboration, which can be configured in two modes: (1) Live replication, where all replicas are active, and operations must be replicated across each replica; and (2) Passive replication, where replicas are deployed but remain inactive until they are needed based on user demand.

A key feature promoted by Hydra is its resource search algorithm, which efficiently locates resources within the decentralized network. Using random ID generation and maximized XOR distance algorithms. It utilizes a distributed hash table called Kademlia [62] and a lookup algorithm to discover any node in the system.

Hydra maintains consistency through a consensus-based algorithm, performing leader elections when necessary. All container resource information is applied to Hydra via a configuration setup in its controller. The system demonstrates robustness in real-world scenarios by ensuring applications continue to function, even in cases where regions or nodes become isolated due to network failures.

Comparison

The solution does not implement a local-first approach, meaning that operations cannot be autonomously executed at each site. However, it does provide a mechanism to ensure that all containers (which act as resources in Hydra) continue to function during network partitions.

Additionally, it supports location-aware deployments, utilizing the available sites dur-

ing such partitions. While resources and applications remain operational at each site during a partition, no new operations can be executed until the partition is resolved.

This solution is not generic, as it is specifically tailored for container-based applications. It supports collaboration by enabling multiple instances of an application to work together via replication. However, it requires significant coordination between sites for operations, as the system must determine the optimal location for each of them. Furthermore, synchronization and consensus processes, especially those involving leader elections among multiple instances, require increased communication. These contribute to the increased synchronization overhead.

Conclusion

Hydra is an approach specifically designed for location aware application geo-distribution. It lacks the necessary generic applicability and does not support autonomous operations at each site. The approach also involves significant synchronization overhead. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.2 Liqo [63]

Liqo is a decentralized solution designed for multi-cluster Kubernetes environments, allowing for the seamless extension of these clusters across geographically distributed locations. It enables dynamic integration of Kubernetes clusters, allowing workloads to be offloaded to remote clusters without requiring any changes to its base architecture or the applications themselves. This approach maintains the use of standard Kubernetes APIs, making it possible to manage offloaded resources as though they were running locally, simplifying management.

The diagram Figure 4.1 illustrates the interaction between a Consumer cluster (the local cluster where the user operates) and a Provider cluster (the remote cluster hosting the resource) in Liqo.

On the Consumer cluster side, the Kubernetes API server and Liqo control plane are responsible for orchestrating and offloading workloads to remote clusters. A Gateway Client facilitates the communication between clusters, forming an overlay network of independent instances. In this Gateway, Kubernetes default communication methods such as Nodeport or LoadBalancer is used.

On the Provider cluster side, the Kubernetes API server and its corresponding Liqo control plane handle the execution of the offloaded workloads.

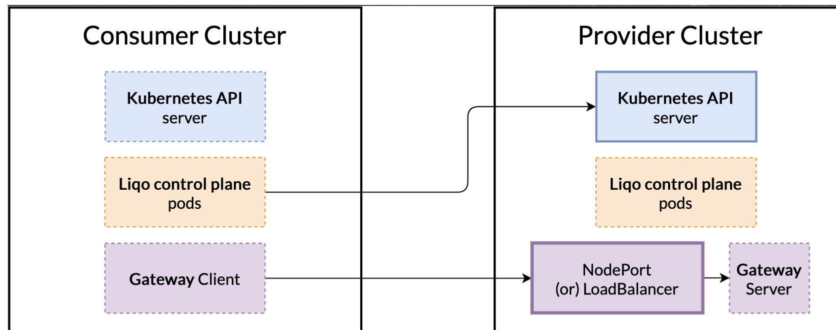

Figure 4.1: Liqo control plane from [63]

Liqo introduces the concept of virtual nodes, which are deployed within the local cluster to represent remote resources. This is achieved using an extended version of the virtual kubelet service [64], allowing workloads to be transparently offloaded to remote clusters while keeping their status synchronized with the local cluster. This enables them to function together as a unified application across multiple clusters.

Resource reflection concept from Liqo, ensures that resources such as Secrets, ConfigMaps, and Services are automatically synchronized between clusters. This automation simplifies multi-cluster management by ensuring that offloaded workloads in remote clusters have access to all necessary resources, without requiring manual intervention. This seamless synchronization is key to maintaining consistent application behavior across clusters.

Comparison

Liqo does not support local-first operations, where actions can be initiated and executed autonomously at a local cluster without depending on communication with a central or remote cluster. This limitation arises because Liqo relies on a leader election and consensus mechanism to ensure consistency and coordination between clusters.

Additionally, Liqo is specifically designed for Kubernetes, which limits its applicability to non-Kubernetes environments. It allows each instance to operate independently during a network partition and synchronize once connectivity is restored. The approach requires significant modifications to deployment configurations. It also modifies the existing Kubelet service by introducing a custom version to create virtual nodes, making it

intrusive to the application.

Conclusion

Liqo is a solution to geo-distribute Kubernetes application while bring the ability to manage multiple instances as one. It is highly intrusive and specific to Kubernetes, lacking the generality required for broader application. Its reliance on coordination for an operation across clusters results in significant synchronization overhead. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.3 OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures [51]

OneEdge is a hybrid control plane that combines autonomous decision-making and scheduling deployments at edge site with a centralized control for management. This approach reduces deployment latency at edge sites while maintaining system-wide coordination, due to the autonomous nature.

For consistency, OneEdge employs a two-phase commit (2PC) protocol, which reconciles the conflicts at the edge (if any) with a central controller.

OneEdge provides a developer interfaces to specify spatial and temporal constraints for applications, ensuring they are deployed in locations that meet specific latency and proximity requirements, an essential feature for real-time, situation-aware applications like connected vehicles or drone navigation.

The system also includes a comprehensive monitoring mechanism that guarantees end-to-end service-level objectives (SLOs) are met. In scenarios of resource scarcity or mobility that might lead to SLO violations, OneEdge dynamically reallocates resources or migrates applications to ensure consistent performance.

Comparison

OneEdge manages network partitions by allowing edge nodes to operate autonomously. However, operations cannot be directly applied to these nodes while they are under network partition. Upon reconnection, OneEdge relies on a central system to synchronize resources that were isolated during the partition. Also, the solution is designed to be adaptable to a wide range of applications, making it a generic solution.

Despite its generic nature, OneEdge requires specific modifications and adaptations to the business logic of each application, making it intrusive for existing ones. Furthermore, in cases of operational conflicts, OneEdge employs a two-phase commit (2PC) consistency protocol, which necessitates extensive coordination between instances. This results in high synchronization overhead.

Conclusion

OneEdge is not suited for a complete autonomous application, due to the reliance on the central control to resolve conflicts. It is highly intrusive and the synchronization overhead is significant, largely due to the locking mechanisms imposed by the two-phase commit protocol. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.4 ENORM: A framework for edge node resource management [65]

ENORM (Edge NOde Resource Management) is a solution aimed at managing edge nodes in fog computing environments. It addresses three primary challenges: provisioning edge nodes for workloads offloaded from the cloud, deploying them effectively, and dynamically managing limited resources on edge nodes through auto-scaling.

The approach includes a dynamic auto-scaling feature that monitors resource availability on edge nodes and adjusts allocations as needed to maintain Quality of Service (QoS), especially under varying workloads. By integrating edge computing with cloud services, ENORM significantly reduces application latency (by 20-80%) and data traffic (by up to 95%), which is crucial for latency-sensitive applications like online gaming and IoT. The design emphasizes integrating edge nodes with cloud servers to enhance performance and reliability in fog computing environments.

Comparison

ENORM follows a Master-Worker architecture, where edge nodes (workers) receive resources from the central cloud (master). It is not a local-first approach, as operations cannot be executed directly on the edge nodes without cloud interaction.

The approach is highly intrusive, requiring modifications to the existing business logic of an application, as demonstrated by examples like Pokemon Go in the article. This

suggests that ENORM lacks the generalizability to easily apply to any application without significant code alterations.

In the event of a network partition, edge nodes can continue functioning, but no new operations or updates can be applied from a local site.

Resource synchronization relies on a centralized system, which reduces the decentralized coordination. This centralized approach can also lead to network congestion at the central site, especially when multiple edge nodes attempt to synchronize at the same time.

Conclusion

ENORM does not support autonomous operations during network partitions. The approach is highly intrusive, requiring significant modifications to an application business logic.

Although synchronization overhead is relatively medium, the reliance on a centralized control node for resource synchronization introduces risks such as network congestion and creates a single point of failure, making the system vulnerable to disruptions. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.5 Shard manager: A generic shard management framework for geo-distributed applications [66]

Shard Manager is an internal framework developed by Facebook, it addresses the challenges of managing sharded applications across geo-distributed environments. This approach is designed to ensure high availability and load balancing of large-scale applications by efficiently managing the distribution and placement of their shards across multiple servers. It supports global shard placement and migration, allowing shards to be replicated and moved across regions.

Shard Manager enables applications to remain highly available even during planned events, such as software upgrades, which are significantly more frequent than unplanned failures, as per the research. It achieves this by gracefully migrating shards to avoid downtime and ensuring that no requests are dropped during shard migration.

By using a constraint solver for near-real-time shard placement, Shard Manager handles complex placement requirements, such as regional preferences, fault-tolerant replication, and resource optimization. Additionally, it provides a programming model that

allows applications to define shard placements and manage their lifecycle efficiently.

Comparison

Shard Manager offers comprehensive collaboration between geo-distributed regions by enabling shard replication and migration across multiple regions. However, this framework is tightly integrated with Facebook's internal infrastructure, making it less applicable to general applications outside of this specific environment.

Shard Manager integrates with existing application without requiring significant modifications to its business logic (as per the article). However, the system requires additional configurations.

During network partitions, Shard Manager ensures that shards remain available through automatic failover and graceful handling with migration. The reliance on global coordination introduces delays, as the framework uses a primary-secondary replication model, with primary shards managing write operations and secondary shards providing redundancy.

Conclusion

While the approach can function on any applications designed for sharding, the requirement for additional configurations makes it somewhat intrusive. It heavily rely on a centralized coordination and has limited flexibility for applications outside of Facebook infrastructure.

Moreover, it incurs substantial synchronization overhead, to ensure that resources remain consistent between primary and secondary shards, it requires frequent communication between instances. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.6 Re-designing Cloud Platforms for Massive Scale using a P2P Architecture [67]

This article explores a peer-to-peer (P2P) architecture to manage applications. This architecture allows cloud applications to efficiently manage massive numbers of resources and users without the need for a centralized controller. The research showcases the application of the P2P model on OpenStack.

One of the key advantages of the proposed solution is its adherence to the principle of minimal intrusion, as it avoids major modifications to the core functionality of existing

cloud management application like OpenStack. The system employs an overlay network for communication between agents across different cloud instances, facilitating efficient resource management and coordination across geographically distributed cloudlets [4]. This makes it particularly suitable for geo-distributed edge and cloud environments.

Each agent in the P2P system independently manages tenant resources and maintains the necessary state information to track resource allocation across cloudlets. The state management system records where resources are provisioned and uses internal IDs that map to the actual resource IDs in the cloudlets. To ensure consistency across multiple agents, this state can be replicated using a consensus-based protocol, ensuring the system maintains a consistent state across geographically distributed applications.

Comparison

This approach employs service-specific proxies to manage communication for each service, effectively creating an overlay network that understands the full application context. While this design minimizes the need for extensive modifications to the core application, it introduces complexity, as developers must create dedicated proxies for each service. This requires a deep understanding of each service, limiting its broader applicability and making the approach non-generic.

The approach relies solely on full copy replication between multiple instances of the application across sites. Handling conflicts during concurrent operations is addressed by a consensus-based approach, though this approach is generic, it prevents a local-first approach to the whole solution.

The approach also incurs significant synchronization overhead since each service proxy need to communicate frequently with others, to ensure consistent operations and synchronization across the entire system.

Conclusion

The approach is non-generic and specifically tailored to OpenStack, which limits its broader applicability to any application. Moreover, the synchronization overhead is high due to the frequent and extensive communication required between different service proxies. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.7 KubeEdge [9]

KubeEdge is an open-source edge computing platform built on top of Kubernetes that extends cloud-native container orchestration to edge environments. It operates with a centralized architecture, where the cloud retains control over the edge nodes. These sites, execute tasks locally while maintaining synchronization with the cloud-based Kubernetes cluster. Although edge nodes can function under network partition, no new operations can be initiated directly from them without cloud involvement.

KubeEdge integrates directly with the Kubernetes API, allowing users to orchestrate resources and manage devices at the edge just as they would in a traditional cloud-based Kubernetes cluster. Synchronization and conflict resolution for any resource operations are handled by Kubernetes RAFT-based consensus mechanism.

KubeEdge employs partial replication to synchronize only the necessary data or configuration from the cloud to the edge nodes, which minimizes bandwidth consumption and avoids full replication.

This architecture is particularly effective for use cases that require low-latency processing, such as IoT applications and real-time data analysis, where bringing compute resources closer to the data source is essential.

Comparison

KubeEdge enables edge nodes to communicate with the cloud for updates and synchronization, but does not promote strong collaboration between the edge nodes themselves. It is built specifically for Kubernetes application, extending its capabilities to edge environments. While it requires minimal changes to the Kubernetes application logic, some configuration adjustments are necessary to manage interactions between edge and cloud resources.

The edge nodes are designed to operate independently during network disconnection, allowing local workloads to continue without disruption. Once connectivity is restored, the nodes resynchronize with the cloud, ensuring the resources are updated. However, the synchronization overhead is relatively high, because edge nodes rely on cloud for conflict resolution.

KubeEdge employs a partial replication strategy, synchronizing only the portion of resource affected by the new updates to specific edge nodes. This selective approach reduces unnecessary data transfer, making it suitable for bandwidth-constrained environments.

Conclusion

KubeEdge is less intrusive (only extra configurations required) and integrates well with Kubernetes, it is specifically designed for this application. This specialization makes it less generic for any application that require support beyond Kubernetes environments. Operations performed on the edge are still controlled by the cloud, which limits the autonomy of the edge nodes.

Although KubeEdge uses partial replication to minimize data transfer, the synchronization overhead is still considerable due to the reliance on the cloud for any operations and conflict resolutions. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.3.8 ShareLatex on the Edge [68]

Traditionally, web applications like ShareLatex are hosted in centralized data centers (core), which lead to high latencies due to the physical distance between users and servers. This study aims to improve application responsiveness by deploying resources closer to users, mitigating latency. The approach leverages individual services of ShareLatex by classifying them according to their statefulness and criticality. Stateless or less critical services are deployed at the edge, while stateful or critical services remain in the core (cloud) for consistency and reliability.

Latency-sensitive services, like text updates and cursor movements, showed improved response times when deployed at the edge. However, operations that require coordination with the core, such as document compilation, experienced increased latencies due to the need for inter-site communication.

Comparison

The approach involve splitting each service within an application (ShareLatex) and placing them at geo-graphically separated sites. While services like real-time editing benefit from edge deployment, critical services like document compilation or notification management still depend on the core, limiting them from being autonomous at the edge.

One of the key benefits is that, this approach does not require changes to the business logic of ShareLatex. Instead, the deployment strategy relies on external tools such as reverse proxies (acting as an orchestrator) for managing services, leaving the source code untouched. However, in the event of a network partition, while certain edge services (e.g.,

Approaches	CT	LF	Collab-T	GN	NI	PH	SO
HYDRA [61]	Decentralized	No	Medium	Low	Medium	Medium	High
Liqo [63]	Decentralized	No	High	Low	Low	Nil	Nil
OneEdge [51]	Hybrid	Yes	Medium	High	Low	Medium	High
ENORM [65]	Centralized	No	Medium	Low	Low	Low	Medium
Re-designing Cloud [67]	Decentralized	No	Medium	Low	Medium	High	High
Kube-edge [9]	Centralized	No	Medium	Low	Medium	Medium	Medium
ShareLatex on Edge [68]	Decentralized	No	low	Low	Low	Medium	Low
Shard Manager [66]	Decentralized	No	Medium	Medium	Medium	Medium	Medium

Table 4.1: Existing Collaboration approaches to geo-distribute an application

stateless ones) can continue to operate independently, stateful services that dependent on core coordination, may be delayed or interrupted.

Coordination between core and edge nodes is required, especially for stateful services that need consistency across locations. This setup creates a less synchronization overhead for services at the edge due to the stateless nature. The system employs partial replication for edge services, where only necessary data is replicated locally, reducing data transfer.

Conclusion

The ShareLatex approach divides services between the cloud and edge based on their characteristics. Implying, a complete standalone instance of the application is not present at each site, limiting the autonomy of each edge node.

Moreover, implementing this method requires deep knowledge of the application to determine how services should be split, reducing its general applicability to any. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

4.4 Summary

This chapter has explored and compared various frameworks designed for geo-distributing applications, with a particular focus to find a possible new collaboration approach that can satisfy Research Question one from Section 3.3.

Each solution was evaluated based on factors such as autonomy, collaboration, generic adaptability, network partition handling, synchronization overhead, and intrusiveness. These solutions demonstrated a variety of approaches to managing distributed resources, synchronization, and collaboration across multiple regions.

While many of these solutions offer promising features such as resource replication, migration, sharding, local-first operations and partial synchronization, all of them fall short in fully meeting our requirements. In particular, the synchronization overhead caused by reliance on centralized control mechanisms, does not align with our vision.

Furthermore, the degree of intrusiveness, to which each solution requires changes to the code in the existing application business logic, varied significantly among them. Solutions like Shard Manager and KubeEdge introduced less intrusion but incurred high synchronization overhead, while Hydra and Liqo were found to be highly intrusive and less generic, limiting their broader applicability.

However, a potential path forward could be a shard-based approach. Sharding, as demonstrated by Shard Manager, provides an alternative to traditional replication approach by partitioning resources across sites. This approach has a potential to reduce synchronization overhead, as it allows individual shards to operate autonomously, minimizing the need for replication. By managing consistency at the shard level, rather than across the entire application, this approach could address some of the key limitation addressed in Research Question 1, as mentioned in Section 3.3.

Chapter 5

Consistency approaches

This section compares consistency approaches in different solutions that geo-distribute an application. The primary goal of this section is to check for a solution that exist for our second research question, as described in Section 3.3. This envisioned approach also need to satisfy our principles, as described in Section 3.1.2. I analyze each approach against a set of comparison points that can potentially lead us to identify a suitable one for my research.

5.1 Design for our approach: Research Question 2

The basic design of Cheops is to provide a middleware that can geo-distribute any application by providing different collaboration models. Each of these collaboration models follow individual approaches towards geo-distribution. Models such as replication requires resources to be consistent across individual geo-distributed sites. The existing approach in Cheops introduced a rollback approach over the RAFT protocol(as discussed in Section 3.1.1). I have highlighted its limitations in Section 3.2.

To address this issue, we are trying to envision an approach that can both satisfy the Cheops principles and the limitations highlighted. In this chapter, we compare various approaches designed for ensuring consistency in a geo-distributed application and try to identify if a suitable one exist to address our question.

5.2 Comparison Points

In this section, I explain the comparison points used to compare various approaches for our envisioned consistency. A collective depiction of all of the surveyed approaches along with these points is illustrated in Table 5.1. Each of these points have been abbreviated to fit better in Table 5.1 and the details are attached along with each explanation. Some abbreviations are not specified in the explanation, they are: Leaderless Consensus (LC),

Physical Time-Based Concurrency Control (PTCC), Causal Transaction (CT), Strong Transaction (ST). The comparison points are:

Local-first (LF): represents the autonomous nature of a solution.

- Yes: An operation can be initiated at any site and will first be executed locally, before being propagated to other remote sites for synchronization.
- No: An operation can or cannot be initiated at any site and will not be executed locally first, it requires additional coordination.

Level Of Consistency (LoC): represents the level of consistency and approach offers.

- Strong Consistency: Guarantees that all nodes see the same data simultaneously after an operation.
- Eventual Consistency: Ensures that, given enough time, all nodes will converge to the same state, but may temporarily hold different data.
- Strong Eventual Consistency: Combines the benefits of both, ensuring that operations are applied in the same order, eventually leading to a consistent state across all nodes.

Consistency Model(CM): represents the consistency model used by the approach. It is dependent on each approach.
Generic (GN): represents the generic nature of a solution. It facilitates managing multiple applications with the same solution.

- High: The solution can be used to geo-distribute any application.
- Medium: The solution can be used to geo-distribute a set of application.
- Low: The solution is specifically designed for an application.

Non-intrusive (NI): represents the non-intrusive nature of a solution.

- High: No change of code in business logic of an application is required to geo-distribute a resource.
- Medium: To geo-distribute a resource, code change is required to create a new configuration for geo-distribution or some specific components such as a service specific broker.

- Low: Need to change the code in the business logic of an application to geo-distribute a resource.

Partition handling (PH): represents if an application can perform during network partition between instances.

- High: Application and all local operations can perform autonomously under network partition between instances.
- Medium: Application and some local operations can perform under network partition between a subset of instances.
- Low: Application can perform in a read-only mode under network partition between instances.

Concurrency handling (CH): represents the concurrency approach used by an approach upon encountering conflict. It is dependent on each approach.

5.3 Existing solutions for Consistency

In this section, I explore existing solutions that could serve as potential candidates for envisioned consistency approach. The presentation of each solution includes an explanation, followed by a comparative analysis with our envisioned approach, and concludes with an evaluation of whether it can be adopted or not.

5.3.1 Rearchitecting Kubernetes for the Edge (RKE) [69]

The proposed approach in the research paper "Rearchitecting Kubernetes for the Edge" involves replacing the strongly consistent datastore (etcd) in Kubernetes with an eventually consistent datastore based on Conflict-Free Replicated Data Types (CRDTs). It is aimed for edge systems, that often have constraints such as limited bandwidth, higher latencies and intermittent connectivity.

One of the key features of this approach is the use of CRDTs, which allow updates to be made independently at multiple sites and automatically resolve conflicts when nodes synchronize. This allows the system to maintain availability even under network partitions, as local nodes can continue operating without waiting for coordination from remote

nodes. Additionally, by adopting lazy synchronization, the system can reduce the overhead involved in achieving consistency across nodes in geo-distributed edge environments. This decentralization moves away from the bottlenecks associated with strongly consistent models, providing a more responsive and resilient orchestration platform at scale.

The unique feature of this approach lies in its ability to reconcile the need for high availability and low latency with eventual consistency. By allowing operations to be executed locally first, it ensures that edge environments, which often suffer from unreliable connections, can still operate autonomously without frequent delays due to synchronization.

Comparison

This approach excels by allowing operations to be initiated and executed locally at an instance before synchronization occurs, ensuring a local-first nature. It allows the application to tolerate temporary divergences between nodes (Kubernetes, by default has a consensus based consistency method) while still ensuring that they converge to a consistent state over time.

The adoption of CRDTs provides a robust mechanism for handling conflicts that arise due to concurrent updates at different nodes. This contrasts with traditional consensus algorithms like RAFT or Paxos, which focus on ensuring strong consistency through majority voting. CRDTs offer a more scalable solution that eliminates the need for coordination, making them ideal for geo-distributed systems.

This approach requires modifications at the datastore level and significant changes to the business logic of applications. It changes the default architecture of Kubernetes, replacing ETCD with another CRDT based data store, this is a significant change to the application. CRDTs require custom data types (making it not generic and intrusive) to ensure proper convergence of multiple instances, meaning that developers need to tailor them into specific use cases.

Conclusion

The reliance on CRDTs for eventual consistency allows for autonomous, local-first operations and better handling of network partitions. This makes it a strong candidate for edge-based deployments where high availability and low latency are critical.

However, the approach is intrusive, as it requires replacing the default etcd component of Kubernetes with another. Moreover, while the approach can perform well in

geo-distributed environments, it is intrusive and not generic. Overall, this approach is a not a suitable candidate for our envisioned approach, due to these limitation.

5.3.2 A Drop-in Middleware for Serializable DB Clustering across Geo-distributed Sites (MSDB) [70]

The approach presented in the paper "A Drop-in Middleware for Serializable DB Cluster across Geo-distributed Sites" offers a solution to clustering databases across geo-distributed environments while maintaining strict serializability. This solution, called Metric, aims to provide a middleware that allows existing applications to achieve geo-distribution without changing their code. The unique aspect of Metric is its ability to work with existing SQL databases, such as MariaDB and PostgreSQL, without requiring changes to the database schema or application logic. It achieves this through a custom JDBC driver, making the transition seamless for applications that rely on these databases.

Metric leverages an Entry Consistent (EC) key-value store to maintain a geo-distributed redo log, which is used to ensure serializability across multiple sites. The redo log tracks the latest record for each table and guarantees that changes made by a transaction are strictly serialized. The EC store allows Metric to manage locks across distributed replicas, ensuring that only one site has access to modify a particular table at a given time. This enables strong consistency, necessary for critical applications where maintaining a single consistent state across all sites is essential.

The system design prioritizes strong consistency and aims to outperform existing solutions by reducing latency and improving throughput across multiple sites. It provides a solution for geo-distributed services that require serializability and are built around traditional relational databases.

Comparison

When comparing this approach to other geo-distributed consistency models, one of the key differentiators is its focus on strong consistency. Unlike models that offer eventual consistency, Metric guarantees that all nodes see the same data once a transaction is committed. This strict serializability is a valuable feature for applications that cannot tolerate temporary inconsistencies, such as financial or critical services.

However, this strong consistency comes at the cost of local-first operations. The Metric approach does not support the local-first model, as it require coordination before a trans-

action can be committed, reducing the feasibility for environments with frequent network disconnections like edge.

Metric approach is intrusive, but applications that already use SQL databases like MariaDB or PostgreSQL can adopt this solution without needing to modify their code. The DB itself requires to change some code to adopt this approach into it. Also, for applications without SQL databases supports, they need to integrate it into their design, making the application change its existing code in the business logic.

Metric is not a generic solution, it works well with supported databases and extending it to others requires additional code. Additionally, it relies on the specific functionality of relational databases, making it less applicable to applications with non-SQL or NoSQL environments.

The system is heavily dependent on coordination, meaning that if a network partition occurs, operations will likely be delayed or blocked until the partition is resolved. This limits its applicability in environments where disconnections are frequent.

Concurrency handling is managed through a lock-based mechanism, specifically using table-level locks in the geo-distributed redo log. This means that before a site can modify a database table, it must acquire a lock for that table across all sites involved. The locking mechanism ensures that only one instance can make changes to a particular table at a time, preventing conflicting writes across geo-distributed application.

Conclusion

Overall, Metric is a good fit for geo-distributed services where consistency is critical, but it may not be the best choice for systems that need high availability during network partitions. The reliance on coordination prevents it from operating autonomously during network partitions, and its lack of generic support for multiple database types.

The approach is intrusive as we need a specific DB (relational database) based approach to geo-distribute an application. Overall, this approach is a not a suitable candidate for our envisioned approach, due to this limitation.

5.3.3 Managing data replication and distribution in the fog with FReD (Fred) [71]

The FReD (Fog Replicated Data) approach focuses on managing data replication and distribution across geo-distributed fog infrastructure. The main goal of FReD is to provide

a flexible, efficient middleware for fog applications to manage their data transparently. FReD uses a keygroup abstraction, where logically related sets of data (key-value pairs) in a resource are grouped together and replicated across instance of fog. Each keygroup can be replicated across multiple nodes in a customizable manner, depending on the requirements of the application.

One of the significant features of FReD is that it allows local-first operations, meaning that operations can be read and written at local nodes before being propagated to other replicas. This optimizes latency for applications that require quick, local access to data. FReD also provides mechanisms for version control and conflict resolution using version vectors (VV), ensuring that eventual consistency is maintained across the geo-distributed system. FReD is highly configurable, allowing applications to define where and how data is replicated in a resource and how long it remains available in different nodes.

Comparison

FReD is designed with eventual consistency as its core consistency model, where high availability and low latency are essential. Instead of requiring all nodes to be synchronized at all times, FReD allows data to be updated locally first and then asynchronously propagated to other nodes, ensuring that the system eventually converges to a consistent state.

FReD allows local nodes to perform reads and writes independently, making it highly suitable for autonomous operation in fog environments. This is a crucial feature when nodes might be geographically distant or subject to unreliable network conditions.

FReD is designed to handle network partitions by allowing operations to proceed locally even when nodes are isolated. Once the partition is resolved, the system will synchronize and resolve any conflicts using version vectors.

It require changes to the application logic, integration with the FReD middleware and its client library for managing data replication and consistency. Applications need to adapt to FReD API and incorporate client-side conflict handling if necessary.

FReD is adaptable to a wide range of applications, not just specific to a particular domain. Its design as a middleware makes it applicable across various fog computing environments, allowing applications to control data replication and distribution as needed.

Conclusion

FReD approach is highly suitable for fog environments where local-first operations and eventual consistency are required. It is a good fit for applications that prioritize low-latency access to data and can tolerate temporary inconsistencies.

But, FReD is intrusive, as it requires integration effort and changing the application business logic. Overall, FReD is a not a suitable candidate for our envisioned approach, due to this limitation.

5.3.4 SessionStore: A Session-Aware Datastore for the Edge (SessionStore) [72]

SessionStore is a session-aware datastore designed specifically for edge and fog computing environments. Its main objective is to ensure session consistency on top of an otherwise eventually consistent data model. In traditional eventually consistent systems, data updates are propagated asynchronously, which works well for many scenarios but can lead to inconsistencies during client interactions with multiple replicas.

SessionStore introduces the concept of session consistency, which provides two guarantees for clients within a session: read-your-writes and monotonic reads. This means that once a client writes data, subsequent reads within the same session will reflect those changes, and the client will always observe a consistent view of the data. This is particularly significant in edge environments, where clients may switch between multiple data centers. SessionStore minimizes data transfer by only synchronizing the relevant session data between replicas when necessary, using a session-aware reconciliation algorithm.

A key feature of SessionStore is its ability to manage client sessions efficiently across distributed replicas, ensuring a smooth transition when a client switches from one replica to another. This is done by tracking the specific data accessed during the session and synchronizing only that data when required, making the system more efficient in terms of latency and bandwidth.

Comparison

SessionStore supports local-first operations ensuring that clients can perform them on their local replica without immediate global synchronization. This reduces latency significantly, especially in edge environments where low-latency operations are critical.

Session consistency is stronger than eventual consistency but more relaxed than strong consistency. This is useful for applications where clients need a consistent view of their own resource but do not require global consistency across all clients. This approach works well for mobile and edge-based applications where the local view of a resource is more important than global consistency.

SessionStore excels in partition handling by allowing operations to proceed locally during network partitions. The system continues to provide consistency for the session when connectivity is restored, and session-specific data is reconciled between the relevant replicas.

SessionStore requires integration with an application to track session data, making it intrusive. Developers must define sessions and track relevant queries for session-aware reconciliation and the core architecture of the application need some changes.

SessionStore is designed for edge and fog environments but may require modifications to work with certain applications, especially those not designed for session-based data access. This makes it specific and not generic, as it works well in specific contexts but might not be a universal solution for all types of applications.

Conclusion

SessionStore is well-suited for environments where local-first operations and high partition tolerance are crucial. However, SessionStore is intrusive, requiring integration effort to manage session data, and it is not generic as it works best in specific application domains. Overall, SessionStore is a not a suitable candidate for our envisioned approach, due to these limitations.

5.3.5 UniStore: A fault-tolerant marriage of causal and strong consistency (UniStore) [73]

UniStore is a geo-distributed data store that combines both causal and strong consistency in a single system. This hybrid approach allows UniStore to flexibly support different types of operations depending on the application requirement.

Causal consistency ensures that causally related operations are seen in the same order by all nodes. This means that if one operation depends on another, the system ensures that everyone observes them in the correct order. For most operations, causal consistency is sufficient and allows UniStore to achieve low-latency updates across different regions,

advantageous for the edge.

UniStore also supports strong transactions, where strict serializability is enforced. These transactions require global coordination, ensuring that operations are processed in a single global order. This is critical for applications where the strongest consistency guarantees are needed, such as financial transactions or inventory management systems. The ability to mix causal and strong consistency makes UniStore unique, providing developers with the flexibility to choose the consistency level appropriate for each operation.

Comparison

UniStore does not provide a pure local-first model for all transactions. Causal transactions can be executed with local-first behavior, meaning they can proceed without waiting for global synchronization. However, strong transactions need global coordination, which limits local autonomy for those operations.

It is designed to work with a variety of applications. Its flexibility to handle both causal and strong transactions makes it suitable for a wide range of use cases, from highly available web applications to mission-critical transactional systems.

In this approach, applications need to differentiate between causal and strong transactions, meaning some integration effort is required to mark certain operations when needed. This may require developers to adjust how they handle different types of operations in their application logic.

UniStore performs well during network partitions for causal transactions. However, strong transactions require synchronization across regions and may be blocked if a partition prevents coordination. This trade-off ensures high availability for most operations while maintaining strong consistency for critical transactions when necessary.

Conclusion

The hybrid approach by UniStore allows applications to benefit from both low-latency causal transactions and globally consistent strong transactions. This makes UniStore a good fit for applications that require a mix of high availability and strict consistency.

However, UniStore is not fully local-first for all operations, as strong transactions require global coordination. It is moderately non-intrusive, requiring some integration effort for operation configuration from developers into the application logic. But, the UniStore approach is a not a suitable candidate for our envisioned approach due to these limitations.

5.3.6 State-Machine Replication for Planet-Scale Systems (SM-RPS) [74]

Atlas is a leaderless State-Machine Replication (SMR) protocol designed for planet-scale systems (any application) that need to provide strong consistency across geo-distributed replicas. The key goal of Atlas is to achieve linearizability, a strict form of consistency, by ensuring that all operations are executed as if they happen in a globally ordered sequence. To do this, Atlas replicates the resource across multiple regions and ensures that all replicas stay synchronized.

One of the unique features of Atlas is its leaderless architecture, which eliminates the need for a single leader node to coordinate all transactions. Instead, Atlas uses quorums to coordinate operations across replicas. For every operation, a majority of the replicas (a quorum) must agree on the order of operations, ensuring that all replicas apply operations in the same sequence. This decentralized approach improves system resilience, especially when dealing with workloads distributed over long distances, such as between different continents.

Atlas follows a dual-path model where operations can take a fast path or a slow path depending on the complexity and conflicts in the transactions. The fast path allows simple, commutative operations to be executed with minimal coordination, while the slow path is used when strict coordination is needed for conflicting operations.

Comparison

Atlas does not allow local execution of transactions. Each transaction, even if initiated at a single location, requires global coordination through quorums, which means the system must synchronize across regions before finalizing the transaction. This global coordination ensures strong consistency but reduces the autonomy of individual nodes.

Atlas does not perform well under prolonged network partitions. When a quorum is not reachable, the system blocks operations until enough replicas are available again to reach consensus. While this guarantees data consistency, it limits the system availability during network failures.

The approach is highly adaptable and can be applied to a variety of applications that require strong consistency, including distributed databases, financial systems, and coordination services. Applications need to integrate with the Atlas protocol to manage operations and coordination across quorums, making it more intrusive. While Atlas removes

Solutions	LF	LoC	CM	NI	GN	PH	CH
RKE [69]	Yes	Strong Eventual	CRDT	Low	Low	High	CRDT
MSDB [70]	No	Strong	Entry	Low	Low	Low	Lock-based
Fred [71]	Yes	Eventual	Causal	Low	High	Medium	Version vector
SessionStore [72]	Yes	Session	Session-Aware	Low	Low	High	Session-Aware
UniStore [73]	No	Causal & Strong	Causal	Medium	High	High for CT, Low for ST	CRDT, 2PC
SMRPS [74]	No	Strong	Consensus	Low	High	Low	LC
CLog [75]	Yes	Causal	Causal	Low	High	High	PTCC

Table 5.1: Existing Consistency approaches to geo-distribute an application

the need for a leader node, it still requires integration with its decentralized architecture, adding some level of complexity.

Conclusion

Atlas is highly suitable for applications that require strong consistency and global coordination, such as financial transactions or systems where strict data integrity is essential. Its leaderless, decentralized design makes it scalable, but it comes with the trade-off of requiring global synchronization for every transaction.

Atlas is not local-first, and it has limited partition handling, as the system blocks operations when a quorum cannot be reached, making it less effective in environments where network partitions are frequent. Also it requires the application to integrate Atlas protocol into its business logic code. Overall, Atlas is a not a suitable candidate for our envisioned approach due to these limitations.

5.3.7 ChronoLog: A Distributed Shared Tiered Log Store with Time-based Data Ordering (CLog) [75]

ChronoLog is a distributed log store designed to handle large-scale, time-ordered events across multiple nodes. The core idea behind it is the use of physical time to order events in a causally consistent manner across a geo-distributed system. Each node in ChronoLog records events locally with timestamps and then uses these timestamps to synchronize logs with other nodes.

A significant feature of ChronoLog is its ability to perform local-first operations. This

means that events can be logged and processed locally without waiting for global synchronization. Later, when the system is ready to merge logs from different nodes, it uses the physical timestamps of events to ensure that their causal order is respected. This approach allows for high availability and low-latency logging operations, which is critical in environments such as IoT.

ChronoLog is designed to be scalable and can handle large volumes of data by distributing logs across nodes while ensuring that events are consistently ordered by time. The use of time-based ordering reduces the need for complex synchronization mechanisms, making it a lightweight and efficient system for geo-distributed environments.

Comparison

ChronoLog allows nodes to log events independently, and the system only synchronizes when necessary, reducing the overhead of constant communication between nodes. It allows local operations to continue during a network partition, ensuring high availability. Once the network is restored, logs are merged based on their timestamps, ensuring that the system remains consistent without losing any data or operations, making it well-suited for environments with unreliable network connections.

However, the time-based logging model is intrusive to the application, as it requires to interact with the API to log and retrieve events. Finally, ChronoLog can be applied to various scenarios, from IoT data logging to scientific applications, making it generic to a wide range of applications.

Conclusion

ChronoLog is a strong candidate for systems that require local-first autonomy, partition tolerance and causal consistency. Its use of physical timestamps allows for efficient, decentralized logging with minimal coordination overhead, which is ideal for geo-distributed systems that need to handle large volumes of data quickly.

However, while ChronoLog is intrusive as it does require some integration with application logic, as developers need to adapt their systems to work with its time-based consistency model. ChronoLog is a not a suitable candidate for our envisioned approach due to this limitation.

5.4 Summary

In this chapter, we conducted a detailed comparison of various approaches to geo-distribute an application with a focus on consistency. The goal was to identify a solution that aligns with our second research question, as outlined in Section 3.3, while also adhering to the principles, described in Section 3.1.2. Each approach was evaluated against several key points, including local-first operations, level of consistency, genericity, non-intrusiveness, partition handling, and concurrency control. The model presented its strengths and weaknesses, with trade-offs in consistency, availability, and partition tolerance based on CAP theorem [76].

From the comparison, we observed that different consistency models cater to specific use cases and constraints. Some approaches, such as ChronoLog and FReD, prioritize local-first operations and high partition tolerance, making them well-suited for applications that require high availability and responsiveness, particularly in edge and fog environments. However, these approaches tend to be more intrusive and require significant integration effort, which reduces their general applicability to a wide range of applications.

On the other hand, solutions like Atlas and UniStore offer strong consistency guarantees, ensuring that operations are globally ordered and serialized. While these models are ideal for applications where strict consistency is critical, they lack autonomy and perform poorly under network partitions, limiting their suitability for decentralized, highly available systems.

Ultimately, none of the surveyed approaches fully satisfies the combination of autonomy, non-intrusiveness, genericity, and partition tolerance required by our envisioned solution. The insights gained from this comparison motivated us to create a new approach that aims to address Research Question two, as described in Section 3.3.

Part III

Contributions

The third part of my PhD thesis presents the key contributions made during my research. I developed three distinct solutions, each designed to address one of the identified research questions. These contributions are organized into four chapters: one chapter outlines the new architecture we developed for the solution, while each of the remaining three chapters focus on resolving a specific research question.

In Chapter 6, I introduce the new Cheops architecture, highlighting the modifications made to enhance its capability to manage resources across geographically distributed application instances. This chapter provides a detailed explanation of the need for these design changes and presents the approach we took to implement them.

In Chapter 7, I propose the shard collaboration mechanism called Cross, which facilitates the partitioning of resources across multiple sites. This chapter addresses the first research question from Section 3.3, introducing Cross as a new collaboration model in Cheops to mitigate the challenges posed by synchronization overhead.

In Chapter 8, I present a novel approach to decoupling consistency from the application logic, ensuring synchronization between geo-distributed instances. This chapter addresses the second research question from Section 3.3, introducing our method for achieving strong eventual consistency without embedding the solution into the application business logic. The approach is non-intrusive, generic enough to be applied to any application, and prioritizes local-first operations.

Finally, Chapter 9 addresses the challenge of maintaining the illusion of a single application across geo-distributed instances during operations. This chapter addresses the third research question from Section 3.3. I demonstrate an approach based on managing the relationships of resources across geo-distributed sites to ensure that operations performed on one instance are reliably reproduced across all instances, preserving the illusion of a unified, single-site application.

Chapter 6

New Cheops Architecture

This chapter introduces the new architecture for Cheops, which is crucial for understanding the contributions described in the subsequent chapters. I will start by presenting the current architecture, followed by an exploration of the significant changes made to Cheops and the rationale behind them. Additionally, this chapter lays the foundation for the validations of my contributions by explaining the experimental setup used in the following chapters.

6.1 Existing Architecture

The Cheops architecture is a modular and microservice-based orchestrator designed to manage geo-distributed applications. Its core functionality revolves around decentralizing control and ensuring communication between different edge sites, creating a service mesh that supports collaboration between deployed individual instances of applications without requiring significant changes to their underlying code, as discussed in detail in Chapter 3.

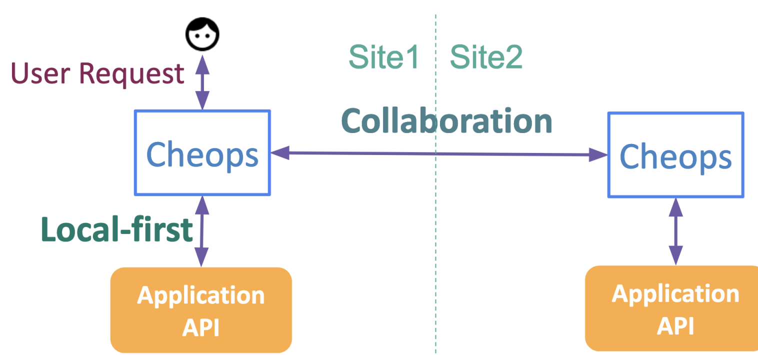

Figure 6.1: Basic Cheops architecture

As illustrated in Figure 6.1, when a user initiates a request at *Site* 1, it is first processed by the local Cheops instance. This local-first approach ensures that any resources or

services available at *Site* 1 are utilized, before involving any external site. The local Cheops instance communicates directly with the Application API, which manages the resource running locally. This helps to reduce latency and ensures quick responses when resources are locally available.

In cases where the required resources or services are not available at the local site, Cheops leverages its service mesh capabilities for multi-site collaboration. This collaboration happens seamlessly between Cheops instances located at different sites (e.g., *Site* 1 and *Site* 2). The Cheops instances communicate with each other to share resources, perform operations, or access remote data.

This inter-site collaboration ensures that the user request is fulfilled even if the needed resources are distributed across multiple locations. In addition, the collaboration mechanism built into Cheops helps in creating a unified view of the application across multiple sites. This allows the application to function as a single coherent system, even though its resources and services are distributed across geographically separated locations. The architecture provides fault tolerance by enabling the application to continue functioning even during network partitions or site failures.

Existing Cheops architecture operates through two primary components: Cheops Core and Cheops Glue, as illustrated in Figure 6.2.

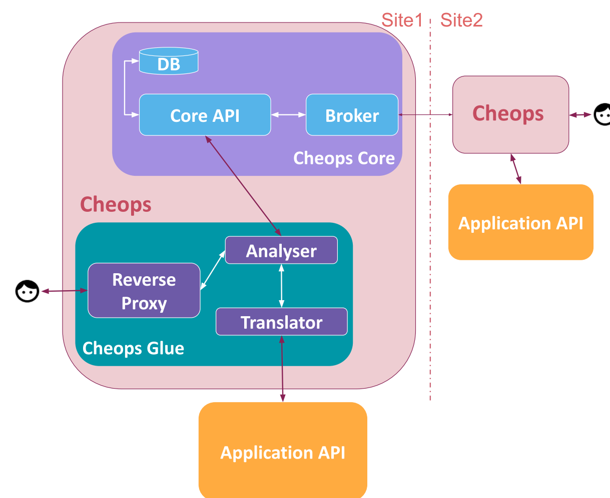

Figure 6.2: Existing Cheops architecture

- **Cheops Core** is the backbone of the system, orchestrating the communication between different instances of Cheops deployed on various sites. It includes the Core API, which acts as the interface between the different services within Cheops and

the deployed applications. The communication module within the Core, ensures that services across different clusters can communicate effectively. This component ensures that deployed applications can collaborate across various sites without altering their core business logic.

- **Cheops Glue** plays a crucial role in adapting the service mesh to the specific APIs of the applications it supports. Since each application (e.g., OpenStack, Kubernetes) has unique requirements for API interaction, Glue helps translate generic Cheops API requests into application-specific API formats. This allows Cheops to remain adaptable while interacting with different application environments. For example, Glue intercepts user requests and translates them using a mechanism called scope-lang, which defines collaboration scopes across sites and allows users to specify where and how their requests should be executed.

The architecture supports replication, sharing, and cross-collaboration between sites, ensuring consistency and seamless communication between applications, spread across multiple geographical locations. For instance, when a request is made to create a resource on one site, Cheops ensures that the request is propagated to other relevant sites, maintaining a unified state across all instances.

The architecture integrates several technologies, including RabbitMQ [77] for communication, ArangoDB [78] for database storage, and HAProxy [79] for request interception.

The core interacts with three essential components:

- **Core API**: This provides the fundamental interface for communication between the Cheops core and application or user. It acts as the central point for routing requests to the right services or nodes across multiple sites. The Core API handles various operations such as managing requests, synchronizing state across sites, and invoking services on remote Cheops instances.

- **Database (ArangoDB)**: This is the database used within Cheops, specifically a NoSQL document-based database. It stores data such as resource information, or metadata with *meta_DB* model, that can be used by the Cheops to manage geo-distribution of applications.

- **Broker (RabbitMQ)**: This serves as the communication bus, built on a peer-to-peer (P2P) Advanced Message Queuing Protocol (AMQP). RabbitMQ enables reliable, asynchronous messaging between different Cheops instances distributed across

sites. It ensures that services running on one site can seamlessly communicate with services on other sites, facilitating collaboration and resource sharing across geographic locations.

Similar to Core, Cheops glue contains three components:

- **Reverse Proxy (HAProxy)**: This is the reverse proxy used for request capture. HAProxy intercepts incoming user requests and forwards them to the analyser. As a reverse proxy, it can be configured to modify an application specific operation from the user, to include scope-lang expressions.
- **Analyser**: The analyser inspects incoming requests and determines how they should be processed. The incoming request will be specific to an application, such as Kubernetes and Openstack. The analyser separates the Cheops part such as scope-lang, site information from the application request.
- **Translator**: The translator converts requests from Cheops into the specific API formats required by the underlying application. This is crucial for applications like OpenStack or Kubernetes, where each has its own set of API calls and operational logic. By translating Cheops requests into application-specific formats, the system ensures compatibility and smooth operation across diverse platforms.

6.1.1 Limitations with the Existing architecture

The existing architecture was a first working version of Cheops that composed of all the principles envisioned in Section 3.1.2. Yet, there were substantial areas in which modification that can be improved, in this subsection, I will detail those:

- **Cheops request**: In the existing architecture, user requests were intercepted through a reverse proxy like HAProxy before being processed by Cheops. A user request consisted of the application request along with the scope-lang expression. For example, to create a Kubernetes pod named foo with replication across both $Site$ 1 and $Site$ 2, the user would issue a command like $kubectl\ create\ pod\ foo\ --scopeSite$ 1&$Site$ 2. This command, while using the original *kubectl* syntax, would require attaching a scope, adding an extra layer of complexity. Since the default Kubernetes CLI is designed to communicate directly with the local application API, Cheops needed to intercept and modify this interaction between the user and the application API.

This necessitated changes in the communication endpoint to redirect the request to Cheops before resuming communication with the application API. The process required modifications to the standard CLI behavior.

- **Database and communication bus**: The initial design relied heavily on ArangoDB for storing resource metadata and managing the distribution of resources across sites. While ArangoDB's NoSQL, document-based model suited the distributed nature of the architecture, it proved to be too complex for managing data synchronization across multiple edge locations. Additionally, the RabbitMQ messaging bus, using AMQP for peer-to-peer communication between sites, introduced significant overhead, especially as the system scaled up to manage more sites. RabbitMQ messaging patterns, though reliable, required careful monitoring and resource allocation, which complicated the overall system.

These limitations, even though it does not affect the working of Cheops, brings substantial areas for us to create a new architecture.

6.2 New Architecture

The new architecture addresses these challenges, it modifies the existing architecture with a new one, as described in Figure 6.3. The base architecture, as described in Figure 6.1, remains the same, with only modification in the components within Cheops.

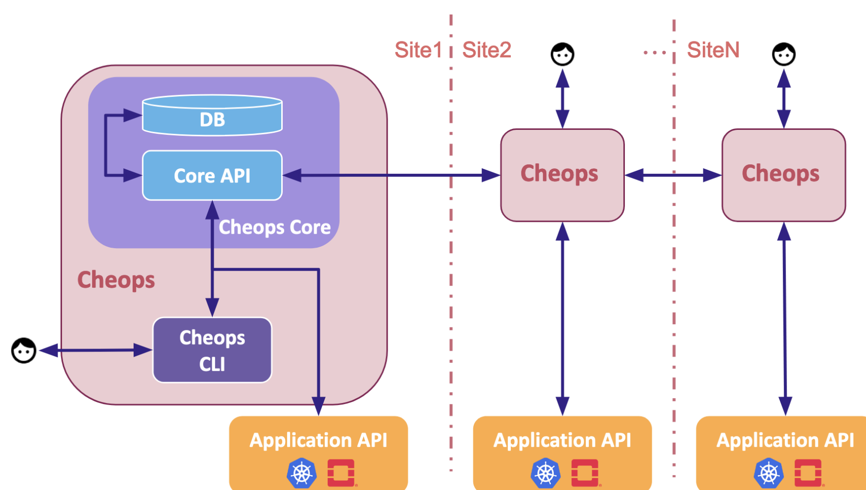

Figure 6.3: New Cheops architecture

In the new architecture, we have introduced a custom CLI, that is designed specifically to handle Cheops requests. This CLI eliminates the need for a reverse proxy and Cheops glue altogether. By allowing users to interface directly with Cheops, it becomes more responsive and adaptable, as the CLI is tailored for Cheops-specific operations, including defining site-specific collaboration and managing geo-distributed resources. This shift provides a more streamlined and user-friendly approach, reducing the overhead previously caused by request interception.

A cheops CLI is illustrated in Figure 6.4, specifically designed to simplify user interactions with geo-distributed applications like Kubernetes or OpenStack. The figure portrays an example with Kubernetes application, sending a request to create a deployment resource in Kubernetes. The Cheops CLI command consists of several key components, including the application command, scope-site definitions, local logic, and resource logic. Together, they allow users to specify operations that need to happen across multiple geo-distributed sites while incorporating the necessary resource management and collaboration requirements. I will detail these components below:

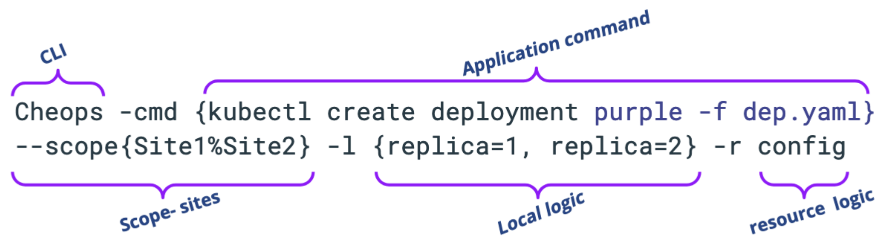

Figure 6.4: New Cheops CLI

- **CLI Command** The command starts with Cheops keyword, indicating that the user is issuing a Cheops-specific command. A user sends the necessary request to geo-distribute an application with this keyword. It invokes the local installation of the Cheops CLI service, to start processing the request.

- **Application Command**: Inside the Cheops command, a typical application-specific command is embedded. In this example, the application is Kubernetes, and the command is $kubectl\ create\ deployment\ purple\ -f\ dep.yaml$. This command would typically be used to create a Kubernetes deployment (purple) using the configuration, outlined in the dep.yaml file. The Cheops CLI does not replace the original application command (e.g., kubectl for Kubernetes); instead, it integrates with it,

nor change its default syntax. By embedding application-specific commands within the Cheops CLI, it ensures compatibility with native commands (making it generic to any application), while extending their functionality to support geo-distribution and application management.

- **Scope Definition**: A key addition that the Cheops CLI brings is the scope-site definition, indicated here as –scopeSite1&Site2. This defines the geographic scope in which the command should be executed. In the example, the command is instructing Cheops to perform a replication collaboration across both Site1 and Site2, enabling geo-distribution between the two sites. The scope-lang expression remains the same, as described in Section 3.1.1 and it simplifies the user task of defining collaboration, ensuring that Cheops handles the intricacies of communicating and synchronizing across distributed locations.

- **Local Logic**: This is a concept, that i will discuss in Chapter 7, through which we integrate the new requirements to extend scope-lang into this version of CLI. *Local Logic* allows to define resources local to an instance when we try to perform Cross collaboration.

- **Resource Logic**: This is a concept, that i will discuss in Chapter 7. *Resource Logic* allows to shard a resource as per the user requirement for Cross collaboration.

Another change, as illustrated in Figure 6.3, is that it replaces ArangoDB with CouchDB [80], a simpler, geo-distribution friendly database that better supports our needs.

The capabilities offered by CouchDB are well-suited to our needs, enabling users to direct operations to specific Cheops agents for resource management while ensuring that changes are eventually propagated across all involved sites. CouchDB can locally register and queue operations, pushing them selectively to relevant agents based on the defined scope (as per the scope-lang).

It can withstand network partitions or agent shutdowns and optimizes network usage by only activating when a new operation is detected, whether initiated by a user or replicated from another agent. Additionally, it checks for existing operations before replication, preventing unnecessary duplication.

The new architecture, removes the communication bus from the earlier architecture Figure 6.2. Reducing or eliminating the reliance on RabbitMQ, makes the new design

adopt a more lightweight communication strategy, utilizing Core API and CouchDB replication model to handle much of the cross-site communication. This results in lower resource consumption, better scalability, and a simpler communication model across sites.

6.3 Experimental setup

We perform experiments that validates each of our new contributions, as described in the later chapters. Here, I will explain the experimental setup for each of them, as they are the same. The experiments itself are different, based on each contribution.

We validate our proposal using the Kubernetes (k8s) application due to the popularity of this container orchestrator. Kubernetes is a software platform consisting of approximately 4 million lines of code, designed to manage the lifecycle of containerized resources within a cluster.

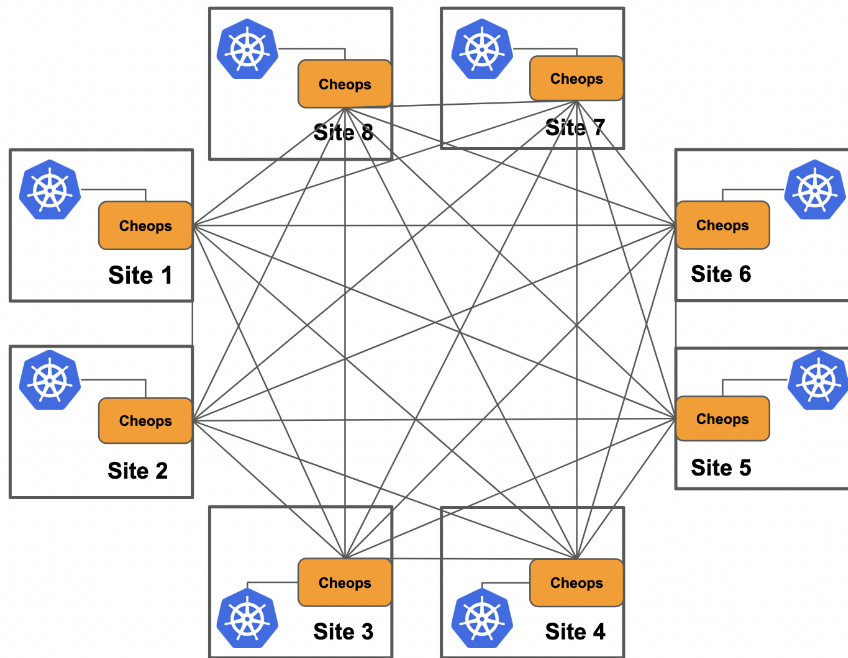

Figure 6.5: Experimental setup with Cheops and Kubernetes

Our experimental setup consists of eight Kubernetes instances located at different sites, emulated on the Grid'5000 testbed [81]. as illustrated in Figure 6.5. We have integrated our proposal into the new Cheops version developed by our STACK team [82].

A Cheops agent is deployed on each cluster, with all agents interconnected. The *Cheops*

software consist of 2500 lines of code in go-lang. Additionally, Scope-lang with the Cheops CLI, used for user interaction, forms another essential part of the Cheops codebase.

We use this setup across all of our experimental validations.

6.4 Summary

This chapter introduces the updated architecture for Cheops, detailing the enhancements made to address the limitations of the original design. It begins by outlining the existing Cheops architecture, which was initially developed as a modular and microservice-based orchestrator for managing geo-distributed applications.

The existing architecture includes core components such as the Cheops Core and Cheops Glue, enabling seamless communication and collaboration between different sites. However, several areas for improvement were identified, specifically related to request handling and data management using ArangoDB and RabbitMQ.

To overcome these challenges, the chapter presents a new architecture designed to simplify operations. Key changes include the introduction of a custom Cheops CLI to handle requests directly, eliminating the need for a reverse proxy (HAProxy) and reducing the complexity of intercepting user commands. The CLI also allows users to define geo-distribution and collaboration scopes with more ease.

The architecture also replaces ArangoDB with CouchDB, a lightweight, distributed-friendly database that better support our geo-distributeion needs across multiple sites. Furthermore, the reliance on RabbitMQ for inter-site communication has been removed, simplifying the communication model and reducing resource consumption.

Finally, the chapter describes the experimental setup used to validate these architectural improvements, focusing on Kubernetes application. The setup involves eight Kubernetes clusters deployed on the Grid'5000 testbed, with Cheops agents running on each site. This setup is used to demonstrate a proof of concept for my contributions in my PhD thesis, described in further chapters.

Chapter 7

Thinking beyond replication, an approach to geo-distribute an application with sharding

---

This chapter introduces my first contribution towards my PhD thesis. It addresses the first question highlighted in Section 3.3, by presenting a new collaboration to geo-distribute any application, called Cross. It is an abstraction over the familiar Sharding concept, i.e., a resource is into smaller shards and geo-distributed across sites. We ensure that the approach follows the Cheops generic and non-intrusive principles. This chapter is an extended version of the article published in ICFEC 2024 conference [17].

7.1 Creating a collaboration with Sharding

This approach is motivated by the limitations from a replication based geo-distribution approach, as mentioned in Section 3.2 and Section 3.3. The major focus is to create an approach to geo-distribute an application, while ensuring less synchronization overhead across sites. We surveyed multiple approaches in Chapter 4. Even though we could not find one that matches our requirements, one in particular stood out, an approach based on sharding.

Shard manager [66], as described in Section 4.3.5 is an approach where they orchestrate individual shards in a geo-distributed application. Sharding involves dividing a software into smaller, more manageable pieces called shards. Unlike replication, where a full copy of resource is maintained across multiple locations, sharding creates pieces of a software that can be deployed at required sites. By doing so, it mitigates the synchronization overhead associated with replication.

Another article that presents a similar shard based application geo-distribution is "Shard scheduler: Object placement and migration in sharded account-based blockchains"

[83]. It presents an approach based on blockchains to geo-distribute shards across multiple sites. One key aspect if this research is the simplicity to manage a shard (piece of software), compared to replication (a complete instance of a software). Also, Shards can be easily migrated or moved around to different locations, unlike replication.

Sharding in general, is a common technique used to distribute large datasets across multiple instances of a database, enabling horizontal scaling and management of large volumes of data. For example: MySQL [84] with Sharding shards the relational DB into smaller tables, MongoDB [85] partitions data across multiple servers , Elasticsearch [86] divides its search index into multiple shards and Google Bigtable [87] uses sharding to distribute data across many nodes in a cluster. Sharding distributes portions of a data across multiple locations, allowing each site to handle more data and traffic by horizontally scaling, rather than duplicating the entire datasets across every instance, like replication.

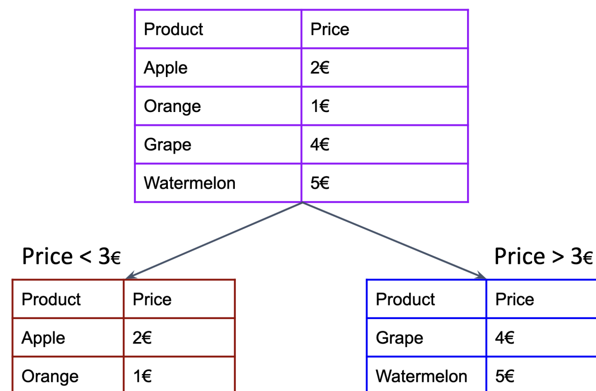

Figure 7.1: An example of a tabular data divided into shards

Figure 7.1 illustrates an example of sharding by dividing a dataset of products and their prices into two smaller subsets based on price. The original table lists four products: Apple (2€), Orange (1€), Grape (4€), and Watermelon (5€). Using the price as the sharding criteria, the dataset is split into two shards. *Shard 1* includes products that are priced below 3€ (Apple and Orange), while *Shard 2* contains products that are priced above 3€ (Grape and Watermelon).

This division, creates two separate datasets from a single one. By doing this distribution of data based on specific criteria, sharding enhances their manageability, as they can now be managed independently and geo-distributed across sites.

7.1.1 Sharding vs Replicating a resource

Figure 7.2 portrays a traditional replication and sharding approach, for a resource R. It highlights how sharding can extend a resource across multiple sites, reducing synchronization overhead and improving scalability compared to full replication.

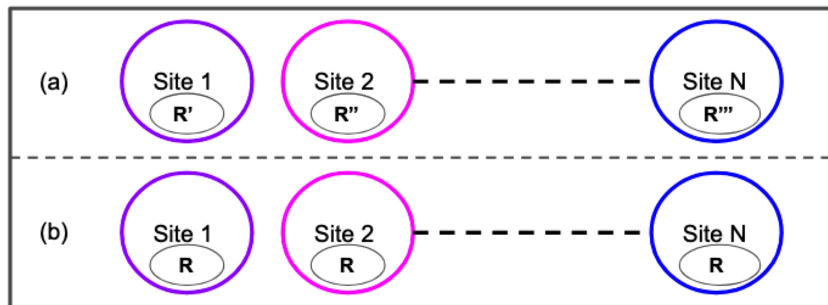

Figure 7.2: (a) Replicating and (b) Sharding a resource

Figure 7.2 (a) Replication: depicts the traditional replication used to manage a geo-distributed application. Resource R is replicated with different sites and a full copy of R is deployed at each of these locations.

While this ensures high availability and fault tolerance, it introduces significant synchronization overhead because any operation from the user must be propagated to each replica to maintain consistency across each site.

Figure 7.2(b) Sharding a resource: illustrates the process of extending shards of a single resource across multiple geo-distributed sites. Resource R is sharded into R', R'', R''', etc., they are individual parts of the same resource located at different sites. Each shard is deployed as an independent resource, collectively forming the original resource R. This approach enhances the scalability of resource management across multiple geographic locations, as it can deploy these individual units at any site and move them as required by the user.

For example, consider a *Set* resource that consist of un-ordered elements {a,b,c,d}. In replication, the resource along with all the elements, {a,b,c,d}, will be replicated across the different sites. An update operation to modify an element a to e, requires to be replicated across all these sites. This causes more synchronization overhead, especially if we consider network partitions and coordinations.

In sharding scenario, the resource *Set*, consisting of the same elements is split into two: *Set 1* and *Set 2*, geo-distributed across *Site 1* and *Site 2* respectively. *Set_1* contains the elements {a,b} and *Set_2* contains {c,d}. For the same update operation, to modify

an element a to e, the request only need to be sent to *Set_1*, less coordination and no replication is required.

7.1.2 Challenges with existing sharding approaches

A challenge to the existing sharding approach is that, it is often tailored to specific application, i.e, it is not generic enough. For example, a sharding strategy that works with a MySQL database will be different for a MongoDB, as one is relational and other is NoSQL. Similarly, sharding on an Openstack application may not be the same as for Kubernetes, as the resources such as Pod and VM have different configurations and composition.

Existing application sharding approaches that I discussed above, Shard manager[66] and Shard scheduler[83], has the same problem. While Shard manager [66] focuses on geo-distributing applications specifically within Facebook's infrastructure, Shard scheduler [83] applies a similar approach but tailors it to a custom-built application that aligns with their unique requirements, invoking intrusiveness. The developers using these approaches are required to change the code in the existing business logic of an application to facilitate sharding.

This reveals a research gap, highlighting the need for a shard-based geo-distribution approach that is both generic and non-intrusive to any application. Cheops, at its core, is designed to promote a generic and non-intrusive method for geo-distributing applications. In my research, I propose a generic and non-intrusive collaboration model based on sharding, which allows the geo-distribution of any application.

Sharding collaboration, entitled, *Cross*, is my first contribution towards my PhD research. A detailed version of this approach has been presented in the ICFEC 2024 conference and is published under the IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid) proceedings [17].

7.2 Cross Collaboration

The Cross collaboration method shards a resource across multiple instances of an application, providing a unified state view (like replication) across sites. The issue observed with the existing sharding approach is that, they require to change the existing base code to support sharding. There are two fundamental questions, that need to be addressed, to

shard a resource with Cross:

- How to partition or shard a resource, such as a pod or a set, into independent, self-sustaining shards (individual resources) in a way that remains generic and non-intrusive.
- How to create an illusion for these shards to single resource for an API request from any site on demand?

These two problems led us to create two important principle, called *Extension* and *aggregation*, that are the core to our Cross solution. The two principles are:

- ***Extension***: This principle shards (divides) a single resource R into R', R'', etc. These shards are independent resources deployed across geo-distributed sites. The extension process need to be non-intrusive and generic to any application.
- ***Aggregation***: Sharding a resource is not enough, this principle aims to create an illusion to unify these individual shards,R', R'', etc., into a single resource R, for a request from any site.

Let us dive into each of these principles:

7.2.1 Extension: Generating new shards

Creating shards from a single resource can follow multiple approaches. The popular approaches in DB sharding are range based [88, 89] (shards a resource based on its range of values), hash based [90] (shards a resource based on its hash values), directory based [91] (shards a resource based on its directories).

Application managers, such as shard manager [66] and shard scheduler [83] shards their resource by having a custom logic built for each resource. Most of the examples demonstrated in these articles have a sharding approach similar to range based.

They have used applications that are tailored to their solution to validate their approach. Existing applications might be incompatible with them or requires intrusive modifications to their codebase to support these approaches.

Cross tries to shard a resource while preserving the non-intrusive and generic principles. The sharding of a resource, happens at the Cheops level, external to an application, making our approach generic and non-intrusive. Lets take a deeper look at this extension process.

At the core of the extension process, we have the Cross database (DB) model, which is responsible for storing the meta data of each shard. The model is structured similar to Cheops replication DB model [14], in which a portion of the data called *meta_identifier* is replicated across all of the involved sites and another, called *local_identifier* consist of values unique to a site (like the resource identification information from a local application) and is not replicated. Cross DB model follows the same design, with two portions of meta data, such that Cheops can easily read both collaboration information. The value inside these two identifiers are different for both models. For Cross DB model, it consist of:

- **Meta_Identifier**: The meta identifier maps any request received by Cheops to a resource spread across all involved sites. This data is replicated across cheops instances, where shards of a resource is deployed. For example, a resource foo, created from *Site* 1 will have a *Meta_Identifier*: *fooSite1 : resource_logic : [Site 1, Site 2 ...]*. Here, the Meta_Identifier value is a identifier created by Cheops for a resource foo. The generation of this identifier involves combining the name of the resource foo with the site from which the create request was initiated. For example, if the resource was created at *Site* 1, the identifier would be $fooSite1$. It also contains information about *resource_logic*, which I will explain later.

- **Local_Identifier**: Cross creates each shard as an individual resource, and the *Local_Identifier* is used to map a shard to the local resource within an application instance. Each *Local_Identifier* is unique to a shard in a site. It contains the name or ID of a resource, which allows Cheops to map any operation on it. For example, the shard information for resource foo at *Site* 1 would be represented as is *foo : fooSite1 : local_logic*. In this case, the *Local_Identifier* would be the name or ID of the resource, and for all involved sites, it would remain the same as the original resource foo and $fooSite1$ corresponds to the *Meta_Identifier* from the resource foo. Additionally, this structure includes details about $local_logic$, which will be explained later.

Local identifiers are specific to each site and identify shards within that site. Meta identifiers, on the other hand, serve as global references that link local identifiers across sites, ensuring a cohesive and unified view of the resource.

Cross model is generic to any resource. This implies, it must be able to capture all potential sharding scenarios. Each resource can have a different sharding requirement, for example, a pod cannot be sharded the same way as a VM, as discussed earlier.

Sharding a set would require splitting the elements onto individual shards, as discussed earlier, but there could be a more complicated resource to shard, such as a list. A list is resource similar to a set, but contains ordered elements.

The challenges associated with sharding a list include determining how to divide ordered elements into separate shards and how to aggregate these elements back into a single, original ordered list. For example, if an ordered list R = [a, b, c, d] is split into two shards $R1$ = [a, b] and $R2$ = [c, d], Cheops must ensure that the original order of the list is maintained when an aggregation request is made. Another key challenge is deciding where to place a new element e (from an insert operation) within the existing shards while preserving the order.

One potential solution is to deploy e within either $R1$ or $R2$, and then perform the aggregation in the original order of R. However, this approach would require the method to be intrusive. A generalized solution cannot be applied in such cases, as specific requirements, such as the ordering of elements within a list, depend on the inherent properties of the individual resource type (in this case, the list), which contradicts the Cheops proposal.

This requires Cheops to introduce a method that can address these specific, resource-dependent requirements while preserving the generic and non-intrusive nature of Cross. To achieve this, we propose an approach to externalize such requirements. This approach involves a program, provided by the user, that defines a set of rules for each resource to ensure these specific requirements are met.

With this method, Cheops is able to intercept and modify user requests to meet any necessary conditions, handling these rules at the API level. To support this functionality, we extend the Cross DB model to include *resource logic* and *local logic*, enabling Cheops to manage the unique requirements of each resource more effectively.

- **Resource logic**: defines how a resource can be sharded and aggregated. It includes a program that contains two functions: extension and aggregation, that are specific to a resource type.
- **Local logic**: Instantiates the resource logic with specific values at each site. These values define each shard.

This approach is akin to Object-Oriented Programming [92]. In this analogy, *resource logic* functions like a class, with the extension and aggregation processes acting as its member methods. Meanwhile, *local logic* corresponds to objects instantiated from this class, following the behavior specified by the member methods. Each shard is initialized

by the *local logic*, and it operates according to the rules defined by the overarching *resource logic*, ensuring that the specific requirements are satisfied during sharding and aggregation across different sites.

Resource logic defines the rules for sharding and aggregating a resource. These rules can be based on the default properties exhibited by a resource type, such as ordering the elements for a list. It includes **extension** and **aggregation** functions tailored to each resource type.

The extension function defines how a resource is divided into individual shards. For example, in the case of a list, the extension function specifies how each shard is created and how the elements should be distributed and ordered across them.

On the other hand, the aggregation function determines how a resource should be reassembled from its individual shards. This process ensures that an illusion of a single, unified resource is maintained, by collecting information from the shards and presenting it as one resource, particularly during operations such as a READ. User need to provide both of them to facilitate Cross collaboration.

Local logic, instantiates the *resource logic* with specific values for each shard at a particular site, as provided by the user. This logic defines a local shard with the necessary resource values or configurations. For example, in the case of a list, to create a shard at *site* 1 with values [a, b], the *local logic* instantiates a new shard with these values using the *resource logic* at *site* 1.

These are combined with the Cross DB model, where *resource logic* is stored with *meta identifiers* and *local logic* with *local identifiers*, as discussed earlier in Section 7.2.1.

Resource logic is replicated at each site along with the $meta_identifier$. When a request to shard a resource foo is initiated across *Site* 1 and *Site* 2, is made, it includes the resource configuration (required for the creation of a resource) and the corresponding Cross DB entry, which contains all these logics.

Before deploying foo at each site, the request is processed and manipulated by Cheops based on these logics, ensuring that instead of a complete resource, only a shard, as described by the user, is deployed. Each operation onto these shares are managed according to the defined rules by the Cross DB model.

An example for *resource logic* and *local logic* for a list resource is described in Code 7.1. It consist of the rule to deploy an element anywhere as defined in the *local logic* and the aggregation function will map all of them, based on a positional value attached to each of them (given during the insert operation for each element). These additional data for a

Code 7.1: Cross Resource_logic definition for a List

```
// Resource_logic:

// Pseudocode for Resource Logic
Aggregation(site S, position P):
    Initialize empty list L
    FOR each site in S:
        Retrieve elements based on position P
        Append retrieved elements to list L
    ENDFOR
    RETURN L

Extension(list L, position P):
    Get sites S from scope-lang
    Divide list L into shards based on position P
    Distribute shards to sites S
```

resource or shard, such as the positional value, is maintained at the Cheops level. Each shard will only have a set of elements as defined by the user.

Code 7.1 consist of a *resource logic* for a list with two functions, aggregation and extension. The ensures that the list can be split into manageable parts and geo-distributed across multiple sites, and later reassembled into a unified view. I detail these functions below:

- **Extension Function**: This function is responsible for sharding a list across individual sites according to the Cross DB model. It assigns a positional value to each element based on its current order. When a new element is to be inserted into the list, the user can specify the value and position of an element using the *local logic*, which is processed within this function. Importantly, the positional values for each element are stored in the Cross DB model, not within the shard itself. Each shard contains only the assigned set of elements, without any embedded positional information.the shard. A shard will consist only of a set of assigned elements.

- **Aggregation Function**: This function reassembles the list from individual shards by following the order determined by the positional values stored in the Cross DB. Using these positional values, the function ensures that the elements from different shards are combined in the correct sequence, recreating the original ordered list as a unified resource.

Local logic, as defined in Code 7.2, illustrates how each shard at an individual site is initialized. It specifies the elements to be inserted into the shard, along with their positional values. This information is processed by Cheops at the required site to create

Code 7.2: Cross Local_logic definition for a List

```
// Local Logic for Site 1
List {Foo, {({a, b, e}, {1, 2, 5})}}

// Local Logic for Site 2
List {Foo, {({c, d}, {3, 4})}}
```

a new shard with these elements. The positional value is sent and stored at the local CrossDB, along with this *local logic* for a shard. The values in local logic are included in the Cheops scope-lang request, ensuring that the shard is correctly initialized and managed at each site according to the specified logic. We discuss about the extension of scope-lang for Cross below.

Extending Scope-Lang DSL for Cross

We have discussed how a resource can be sharded across geo-distributed sites, but how can a user define these shards? Scope-Lang, a Domain Specific Language (DSL), is designed to facilitate the management of geo-distributed resources within Cheops, as outlined in Section 3.1.1. It already supports Sharing and Replication collaborations, and we extend the same to support Cross.

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

$$
\mathcal{R}[\![s : App_i]\!] = s_i
$$

$$
\mathcal{R}[\![s : Loc\%Loc']\!] = \mathcal{R}[\![s : Loc]\!] \text{ extended to } \mathcal{R}[\![s : Loc']\!]
$$

Figure 7.3: Scope-lang expression syntax for Cross collaboration model

The expression of Scope-lang is defined in Figure 7.3, it provides a structured syntax for describing how services in different instances of a geo-distributed application collaborate.

The scope, denoted as σ, contains location information ($Loc\&Loc'$) that is mapped to a service (s) in an application. For instance, in Kubernetes, s represents the Kube-API server service, and $Loc\&Loc'$ refer to the individual sites where the application is geo-distributed. We define the Scope-Lang expression for the Cross collaboration with the operator symbol %.

A basic example of this expression, used with any application to manage a resource R, is as follows:

`application create R --scope {`$Site\ 1$ `%` $Site\ 2$`}`

It is integrated with the default application API, extending it with additional information to facilitate geo-distribution. This allows users to easily specify the geo-distribution requirements for sharding across different sites while continuing to use the familiar application API, without requiring changes to the underlying application code.

Scope-lang provides a method for inputting a request to Cheops, and we extend this by including both *resource logic* and *local logic* along with the request. *Resource logic* is attached as a declarative program file that accompanies the request, while *local logic* is included as a parameter alongside the scope. The complete Scope-Lang expression for Cross, incorporating both logics, will be as follows:

```
application create R --scope {Site 1 % Site 2}
        --resource_logic rl.go
  --local_logic{Site 1:values, Site 2:values}
```

This expression ensures that Cheops can manage the shards with these logics across geo-distributed sites.

Resource logic in the Scope-Lang expression can be provided either with every request or only during the initial CREATE request (the first request for a resource). Cheops will store this *resource logic* at each of the involved sites, along with the corresponding $meta_identifier$ in the Cross DB model, as discussed in Section 7.2.1. This ensures that the *resource logic* is consistently available at all sites, allowing subsequent requests to reference the stored logic without needing to redefine it for each operation.

For instance, consider a Pod foo that needs to be sharded across two sites, $Site$ 1 and $Site$ 2. The Pod consists of two containers: $c1$ and $c2$. The user defines a resource logic to split the Pod based on these containers, where one container is deployed at each site. The sharding happens based on splitting a Pod with two images into two individual pods.

The *resource logic* and *local logic* define the rules for both extending and aggregating the Pod, including the code for splitting and reassembling foo based on the containers. In this case, Pod foo will be extended into two shards: foo' with $c1$ deployed at $Site$ 1 and foo' with $c2$ deployed at $Site$ 2. Both shards are managed by Cheops, and during aggregation, these shards are combined to recreate Pod foo, maintaining a unified state across the two sites.

The request with Scope-lang for Kubernetes application the same will be: `kubectl`

```
create pod foo --scope {a: Site 1 & Site 2} --resource_logic rl.go --local_logic
{Site 1:{container: c1}, Site 2:{container: c2}} .
```

Extension workflow

Here, I explain a workflow for the Extension process with a resource *foo*, as illustrated in Figure 7.4. A CREATE request is initiated from *Site* 1 to create a resource *foo*. This request triggers the resource creation and initiates the Cross collaboration process across the geo-distributed sites.

The resource *foo*, which is of set data type, needs to be extended between *Site* 1 and *Site* 2. At *Site* 1, the resource should be initialized with elements [a,b], while at *Site* 2, it should be initialized with [c,d]. In this workflow, we do not consider network partition between sites.

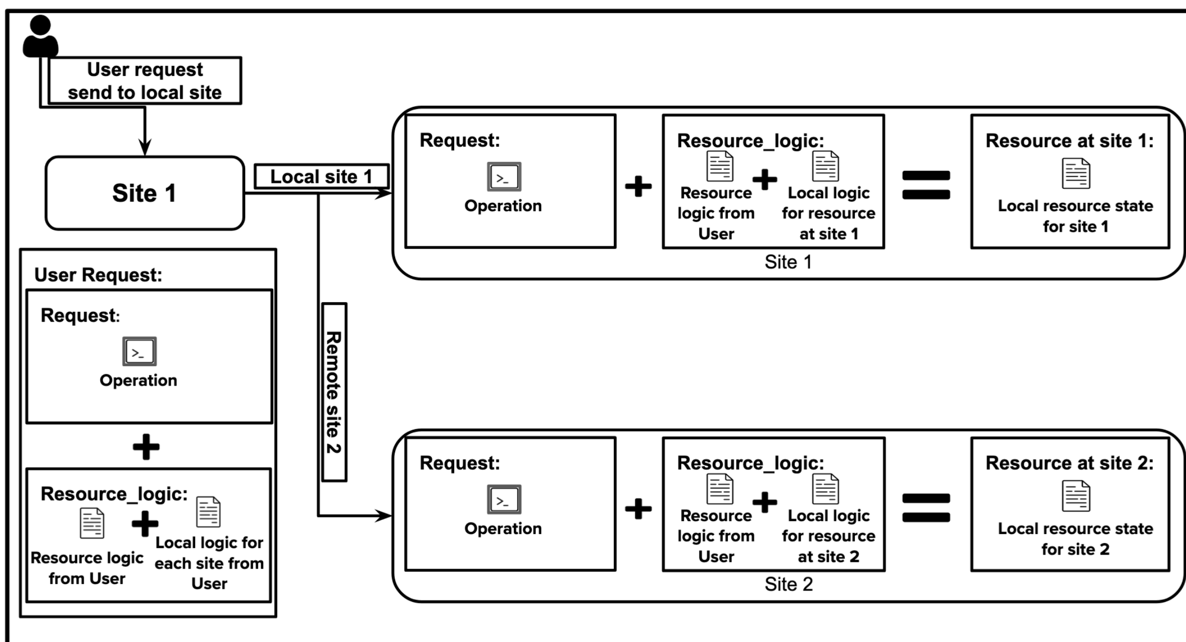

Figure 7.4: Extension workflow in Cross collaboration model

- **Step 1: Initiate Request**
  - A CREATE request is send to *Site* 1 Cheops to create a resource *foo*, extended across *Site* 1 and *Site* 2.
  - It contains the resource definition create the resource, along with Scope-lang to specify the geo-distribution requirements to Cheops.

  - The Scope-lang includes resource logic, illustrated in Code 7.1, representing the way to shard a resource foo. It also includes the local logic for each site, illustrated in Code 7.2, representing the value to initiate each shard.

- **Step 2: Propagating request to the involved sites**
  - Cheops sends a copy of the CREATE request along with the Scope-lang to each of the site included in the request.
  - Cheops at *Site* 1, instantiates a local shard with resource logic and local logic value assigned to the site. It creates a shard for foo with values [a,b] and responds with a 200 *success* REST API code.
  - Cheops at *Site* 2, does the same and instantiates a local shard with resource logic and local logic values assigned to the site. It creates a shard for foo with values [c,d] and responds with a 200 *success* REST API code.

This workflow portrays the extension process of a resource foo, the cross collaboration does not end in only creating individual shards, it has a procedure to aggregate them to create an illusion of a single resource foo. I will explain in detail this second principle called aggregation.

7.2.2 Aggregation: a consolidated view of the shards

Aggregation refers to the process of combining geo-distributed shards into a unified resource view. Despite being distributed across multiple sites, Cheops maintains the illusion of a single, unified resource, as illustrated in Figure 7.5. This is accomplished through a coordination mechanism that collects and aggregates responses for a request from the individual shards.

For example, consider a scenario where a list resource R is distributed across two sites, *Site* 1 and *Site* 2. The original list contains elements [a, b, c, d, e], which have been divided into individual shards, R' = [a, b, e] at positions 1, 2, and 5 and R'' = [c, d] at positions 3 and 4.

A READ operation is initiated by the user from *Site* 1, the request is sent to all of the involved shards and the responses are collected back at *Site* 1. The aggregation function, as described in Code 7.1 iterates over each of the responses. From each response, it retrieves elements and their positional value P, Cheops orders them based on P and returns an aggregated list R, which combines all the elements in their correct order.

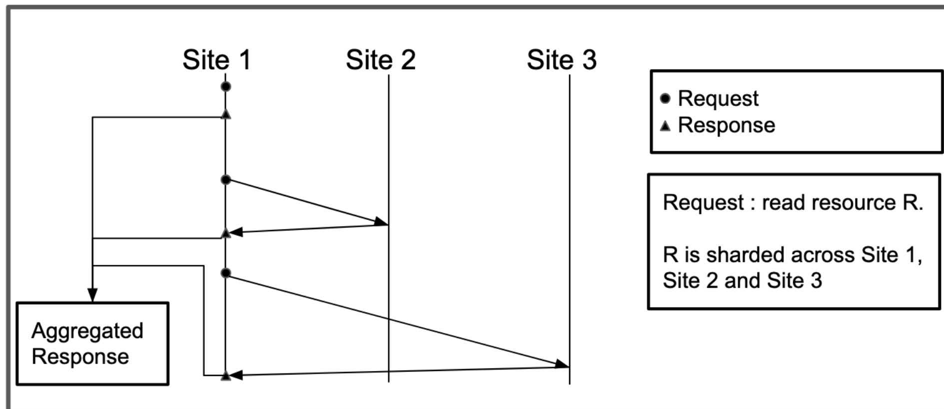

Figure 7.5: Aggregation response in Cross collaboration model

Aggregation performs optimally when all networks are fully operational. However, there may be scenarios where one or more of the participating sites experience network partition. In such cases, it is essential to implement a mechanism that returns a partial response from the active sites, while simultaneously reporting errors from the inactive ones, to the user. The following section outlines an approach to address this challenge.

Handling network partitions at the API level

Network partitions occur when there is a loss of connectivity between different sites in a geo-distributed application. Cheops ensures that an application at each site continues to perform local operations, even during a network partition.

But due to network partition, the operation propagated to a remote site can result in the request being lost or time out. To avoid loss of requests, informed decisions are required from individual sites. Here, we introduce a new approach called **partial error**. It integrates with Cheops to indicate whether a particular site is unreachable or has failed to perform an operation with a detailed response.

It gathers individual responses from each site using a synchronous communication method for every request. In the event of a network partition, the response will include the error message that Cheops received from the request sent to the affected site. This ensures that any communication failures or errors are captured and reported back to the user. This specific error message can help the user to decide on how to resolve the issue.

Partial error handling is designed to capture any errors that may occur due to either network partitions or application-specific issues. To achieve this, each request sent to the involved sites is isolated by the Cheops agent from one another, ensuring that if an error

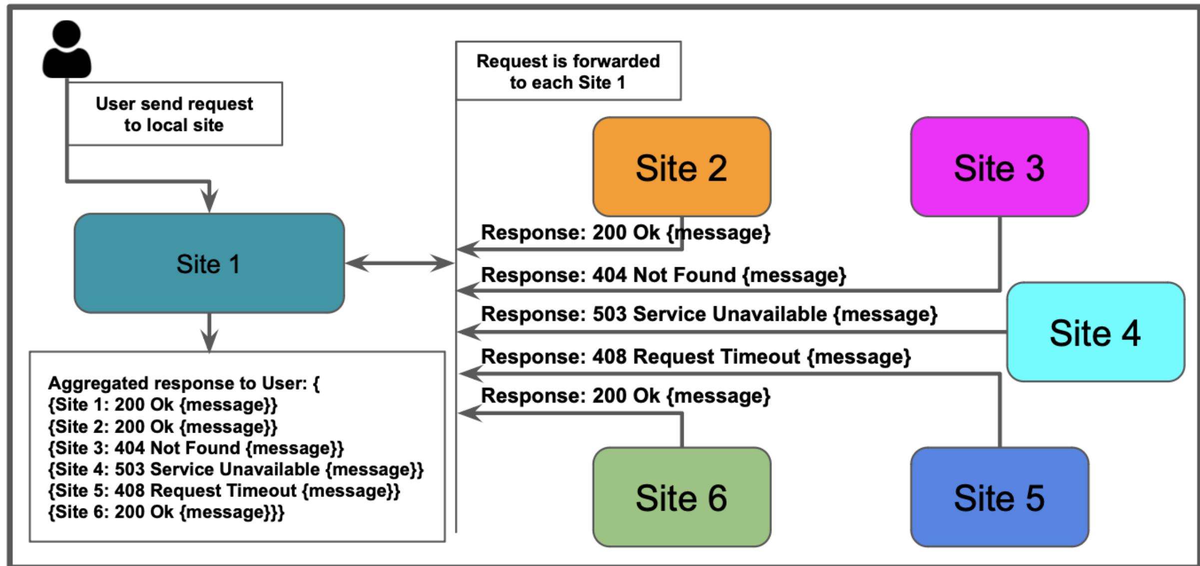

Figure 7.6: Partial error mechanism for Cross collaboration model

occurs at one location, the site can be identified.

Figure 7.6 illustrates an example for our partial error concept, a request is initiated from *Site* 1 to several other sites. Cheops at *Site* 1, replicated this request and send separate ones to each site and they respond individually back to *Site* 1. At *Site* 1, these responses, both successful and error messages from each site, are aggregated. The combined response is then returned back to the user.

A timeout is enforced for each request to prevent it from waiting indefinitely, as each request is dependent on a response. In the event of a timeout, an error message is returned to the user. This initial version of the approach is less automated, but there is potential for further automation with this information in future. Let us look at the workflow for aggregating a resource:

Aggregation Workflow

We present the workflow for a READ request. A READ request is sent from *Site* 1 to resource foo. The resource foo is a *Set* resource and is already split into two shards: at *Site* 1 with values [a,b] and *Site* 2 with values [c,d].

`Scenario 1: All the sites are active and no network partition.`

- **Step 1: Initiate Request**
  - A READ request for the resource foo is initiated by the user at *Site* 1 to Cheops.
  - Cheops checks the local Cross DB model and identifies that foo is spread across *Site* 1 and *Site* 2.
- **Step 2: Propagating request to the involved sites**
  - Cheops sends individual READ requests to each of the Cheops instances at the respective resource sites.
  - Cheops from *Site* 1 responds with a 200 *success* (the REST API code) and returns the value from the local shard: [a,b].
  - Cheops from *Site* 2 responds with a 200 *success* (the REST API code) and returns the value from the local shard: [c,d].
- **Step 3: Aggregate Responses and return to user**
  - *Site* 1 aggregates both the responses, with the help of the function defined in *resource logic* (as described in Code 7.1) and presents: *Site* 1: [a,b], *Site* 2: [c,d].
  - The final aggregated response, containing both successful responses, is sent back to the user.

`Scenario 2:` *Site* 1 `is available, but` *Site* 2 `is under network partition.`

- **Step 1: Initiate Request**
  - A READ request for the resource foo is initiated by the user at *Site* 1 to Cheops.
  - Cheops checks the local Cross DB model and identifies that foo is spread across *Site* 1 and *Site* 2.

- **Step 2: Propagating request to the involved sites**
  - Cheops sends individual READ requests to each of the Cheops instances at the respective sites.
  - Cheops from *Site* 1 responds with a 200 *success* (the REST API code) and returns the value from the local shard: [a,b].
  - *Site* 2 is currently facing a network partition, the site is unreachable, the request responds with a 504 Gateway Timeout and returns the message: Unable to retrieve resource from Site 2 due to network partition.
- **Step 3: Aggregate Responses and return to user**
  - *Site* 1 aggregates both the responses, with the help of the function defined in *resource logic* (as described in Code 7.1) and presents: *Site* 1: [a,b], *Site* 2: Request timeout
  - The final aggregated response, containing both successful and failed responses, is sent back to the user.

In this case, Cheops is able to recognize that the resource is geo-distributed using the Cross collaboration model. Within Cross collaboration, the user has the option to define a custom aggregation method, which will guide the aggregation process. It is defined with the *resource logic*, as defined in Code 7.1 for a *List* resource. This ensures that even in the presence of partial errors, Cheops knows how to aggregate responses from available shards based on the specified logic.

Cross collaboration is a combination of both Extension and Aggregation workflows. It facilitates a sharding approach to geo-distribute any application, but is a resource completely shardable?

7.3 Can a resource be completely sharded?

We have explained our approach to shard a resource within an application; however, not all components of a resource are easily shardable. Certain elements, due to their inherent nature, must be replicated across all instances in a geo-distributed application. This creates a scenario where shardable and non-shardable components coexist within the same resource.

Shardable Attributes: These are components of a resource that can be sharded across different sites with minimal synchronization required between them. For instance, in a *set* resource, individual elements can be distributed across different shards, as explained earlier in Section 7.1.1. Each shard is responsible for managing a subset of elements within the original *set*.

Non-Shardable Attributes: These are attributes that must remain consistent across all instances because they are essential to the identity and functionality of a resource. Non-shardable attributes, such as the *name* or *metadata* of a *set* resource, cannot be distributed or divided across multiple shards, as they need to be uniform across all instances to preserve the original resource integrity.

For example, a user operation to add an element x to the set *foo*, `application add -resource foo -element x`, references the resource by its *name*, foo. If the *name* differs across shards, Cheops would have to modify the operation at each site to account for the different local names, making the process more intrusive. Therefore, attributes like the *name* must be consistently replicated across all shards to ensure uniformity and reduce operational complexity.

Furthermore, if the name foo is updated to bar, this change must be propagated to all shards. This implies that any modification to non-shardable attributes, such as the name, must be replicated and synchronized across all shards to maintain consistency and prevent operational conflicts.

By default, Cross, with the help of resource logic, allows users to specify which elements should be sharded. If an element is not explicitly defined in the resource logic as shardable, it is automatically replicated across all instances. This ensures that only the necessary components are sharded, while the remaining elements are consistently available across all sites without additional configuration.

7.4 Validation

The goal of this section is to demonstrate a proof-of-concept for Cross collaboration within a geo-distributed Kubernetes application. Additionally, we outline a case study showcasing how our solution can be extended to other applications, such as ShareLaTeX, which orchestrates multiple services to manage a LaTeX document collaboratively.

We utilize the experimental setup described in Section 6.3, involving multiple independent Kubernetes clusters. The validation focuses on two Kubernetes resources: namespace

and deployment. I will provide detailed explanations of the approach below.

Kubernetes Namespace

A *Namespace* (abbreviated as NS) is an abstraction in Kubernetes to isolate a set of resources from others. In this experiment, a foo NS is created with a pod a in *Site* 1. foo is extended from *Site* 2 to *Site* 8 with cross collaboration. We use Cheops CLI with kubectl along with scope-lang to define such a distribution from *Site* 1 as `cheops --cmd kubectl create ns foo --scope {Site 1 % Site 2 % Site 3 % Site 4 % Site 5 % Site 6 % Site 7 % Site 8}`.

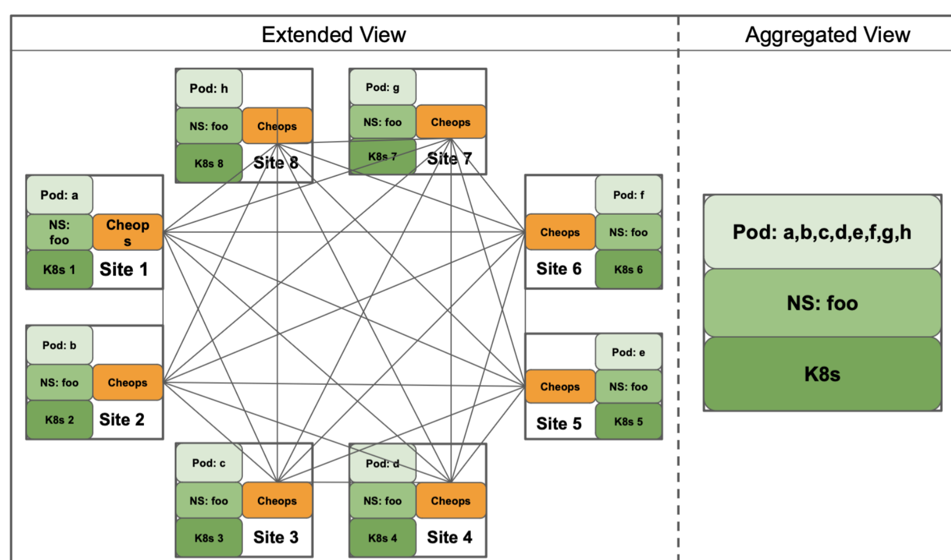

Figure 7.7: Cross Namespace: Extended & Aggregated View

The notation % represents the *Cross* collaboration in this Cheops operation. A Pod, labeled from a to h, is deployed across *Site* 1 through *Site* 8, respectively. Each site hosts a distinct Pod corresponding to its label, creating a geo-distributed setup across the eight locations.

The request to deploy pod a from *Site* 1 will be `cheops --cmd kubectl create pod a -f poda.yaml --scope {Site 1}`, where we use scope-lang to specify the site of the pod in the Cross namespace. Similarly, all the pods a to h across *Site* 1 till *Site* 8 are deployed.

For the Cross namespace foo, configuration elements such as the name, labels, and similar metadata are replicated across all sites since they are non-shardable elements. However, elements like pods within the namespace are not replicated; instead, they are distributed across multiple sites, ensuring each site manages its own set of pods.

Code 7.3: Cross resource_logic definition for a Deployment

```
// Resource_logic:

Aggregation(sites S, resource R, configuration C):
// Concatenate non-sharded elements from all sites
FOR{i=1 TO len(S)}
        R_S = R_S + replica value at i
ENDFOR
Replace replica field with sum_of_replicas in C
RETURN (C)

Extension(resource R, value V):
Retrieve Configuration C from R and Sites S from Cross_DB
C_RS = Change replica attribute of C based on V
Apply C_RS onto Site S
```

The first part in Figure 7.7 represents an extended view of the resource foo distributed across multiple sites. It shows the distribution of individual pods at each site. The other part presents an aggregated view of the resource foo, giving an illusion of a single Namespace foo across each site.

No additional code is needed to geo-distribute a Namespace, as there is no explicit division of elements happening within it. The pods are deployed individually using Cheops onto the Namespace, so no extra mechanism is required to split them.

For aggregation, the responses from multiple sites need to be combined. However, no extra code in the *resource logic* is necessary, as Cheops performs operations (such as a READ) individually at each site and aggregates the results, as discussed in Section 8.5.

Kubernetes Deployment

A *Deployment* is another k8s resource that creates and manages a number of pods as replicas, within a single cluster, based on a configuration. We try to geo-distribute a deployment by sharding the *replica* attribute in the configuration, such that the user can choose how many replicas are to be instantiated at each site.

There exist attributes that belong to non-shardable, such as deployment name, labels, container names, etc. By default Cross replicates these attributes, which are not specified in the *resource_logic*.

Our scenario consist of a deployment *bar*, requesting to create five replicated pods across *Site* 1 and *Site* 2. An operation is performed to create 2 *pods* on *Site* 1 and 3 *pods* on *Site* 2 with command `Cheops --cmd kubectl create deployment foo --scope {Site 1%Site 2} --local_logic {Site 1:{replica = 2},Site 2: {replica = 3}}`.

Code 7.3 depicts the *resource_logic* for this operation. The only sharded attribute

in the *resource_logic*, is the *replica* key from the configuration of the deployment, given by the user. It can be instantiated by the *local_logic* from the request, by `local_logic = {Site 1:{replica: 2},Site 2: {replica: 3}}`, the total number of replicas deployed is the sum from both sites, resulting in 5. The aggregation and extension function defines how to handle the distribution of the resource and the *local_logic* instantiates *resource_logic* with values for sharding the resource at each site.

Handling Network partition under Cross collaboration

We present an experiment to portray the behavior of both sharded and non-sharded elements, in Cross collaboration, under network partition. We target a Deployment resource, that consist of shardable and non-shardable elements.

To test the non-sharded elements, we simulated it by disconnecting the Cheops instance at *Site* 3 from others. We tried to update a *label* (an attribute in the configuration of the deployment, consisting of a key-value pair), from *Site* 3 and concurrently from other sites. One of the replicated labels had an initial value as *colour:brown* across all sites.

After disconnection, we updated the value to *colour:blue* at *Site* 1 and to *colour:red*, at *Site* 3. For the *Site* 1 update, Cheops received a result with partial error containing a REST code 200 (success) from all sites except *Site*3, where this specific site resulted in 404 (No connection). Similarly, for the update from *Site* 3, 404 (No connection) REST code was returned from all sites except *Site* 3, where this specific site resulted in 200 (success).

An operation performed during a network partition on a shardable element, needs to be propagated to all the sites eventually. Once the network came back, Cheops at *Site* 1 and *Site* 3, will push the operations to other sites. This will create a concurrent consistency issue, which we solve in Chapter 8, which is dedicated to this approach.

For a shardable element in the deployment configuration, we use the *replica* attribute, as presented in Code 7.3. Our scenario consist of a deployment *bar* consisting of five replicated pods across *Site* 1 (2 pods) and *Site* 2 (3 pods). We induce a network partition at *Site* 2, severing the communication between Cheops instances.

A local operation is applied from *Site* 2, to update the local deployment *replica* attribute to 5 pods. The operation is: `Cheops --cmd kubectl update deployment foo --scope { Site 2} --local_logic {Site 2: {replica = 5}}`

The operation will result in a success at the local Cheops instance of *Site* 2. Since

this update is applied to the local shard, there is no need for synchronization or remote propagation of this request to another site. The total number of replicas across *Site* 1 and *Site* 2 will become 7.

7.4.1 Use-case: Extending Cross to Sharelatex

We try to create a study to extend the cross collaboration model to Sharelatex [68] application. This analysis focuses on geo-distributing the application, we did not perform the actual experiments for this.

Sharelatex, is a web-based collaborative platform for working with LaTeX, a typesetting system commonly used for scientific and mathematical documents. It allows users to create LaTeX documents, enabling real-time online collaborations and rich text editing.

It is composed of multiple services in a single Site as explained in the research [68], which introduces a proxy approach towards geo-distributing these services. The proposed approach is too intrusive to the application and it seized operating during network disconnections, as a complete instance of an application is not available at each site.

In our proposal, an entire instance of Sharelatex application is deployed at each of the involved sites. LaTeX *project*, is a resource in Sharelatex, that isolates all the files related to a single research document. We consider all the configuration elements related to a project such as name, labels etc as non-shardable resource, since sharding these identifiers with local values does not bring any added value and can create issues while performing an operation. The sub-resources *files* inside a project, can be considered as shardable elements, we shard them and keep a portion of these files at each location.

Consider a LaTeX project foo that has different files *main.tex, image1.png, image2-.png, intro.tex, conclusion.tex.* Extending and geo-distributing foo with Cross across *site* 1 and *site* 2 creates sub-resources foo' and foo'' respectively. In our distribution, foo' consists of *main.tex, image1.png, image2.png* sub-resources and foo'' consist of *intro.tex, conclusion.tex.*

Such a division can be facilitated by the extension principle, involving *local_logic* and *resource_logic*. Cheops will ensure that operations from the user will get the illusion of a single foo across *site* 1 and *site* 2. While compiling a project, Cheops ensures that all these files are connected by the aggregation function defined in the *resource_logic*. If any sites are disconnected, Cheops returns a partial error to the user, with the locations that cannot be accessed.

This proposal, proves that the Cross collaboration in Cheops can be adapted to any application.

7.5 Summary

In this chapter, we introduced Cross, a novel collaboration method designed to geo-distribute applications by leveraging the concept of sharding. This contribution addresses the limitations of replication-based geo-distribution strategies, which often involve high synchronization overhead across distributed sites. Sharding, in contrast, allows resources to be divided into manageable fragments, reducing the need for frequent synchronization.

We explored how Cross abstracts the traditional sharding technique into a generic, non-intrusive solution that can be applied to a wide range of applications. Unlike existing sharding approaches that are typically tailored to specific platforms, Cross remains application-agnostic, enabling it to support diverse systems like Kubernetes, Openstack, and Sharelatex. This chapter also presented two core principles of the Cross approach: Extension and Aggregation, providing mechanisms for dividing resources into shards and reassembling them into a unified view as needed.

The experimental validation demonstrated how Cross can be implemented effectively, using Kubernetes application. The case study of Sharelatex further illustrated how this collaboration model can be adapted to various applications. Additionally, we discussed the handling of network partitions through a partial error mechanism, ensuring that geo-distributed resources remain operational even when communication between sites is disrupted.

This sharding-based approach offers a new way to geo-distribute resources across geographically separated locations, maintaining the illusion of a unified system while minimizing overhead.

Chapter 8

Externalizing consistency between geo-distributed Instances

This chapter presents our approach to ensure consistency between geo-distributed application instances. It contributes as my second contribution to the thesis. In Section 3.2, I discussed the limitations of the existing Cheops consistency approach with Raft. Based on these limitations, we explored several solutions for maintaining consistency across geo-distributed applications, as detailed in Chapter 5. However, none of these solutions fully align with our principles, as outlined in Section 3.1.2, specifically the need for a generic and non-intrusive method to ensure consistency. In this Chapter, I will introduce our consistency approach, which adheres to these principles.

8.1 An External Approach Towards Consistency

Existing synchronization methods in geo-distributed applications, as described in Chapter 5, often embed consistency mechanisms within the application logic, making them intrusive and requiring significant code changes. These methods frequently rely on consensus protocols like Paxos or RAFT [93], which can stall operations during network partitions, thus reducing availability. Alternatives such as CRDTs [50] allow for independent local operations and ensure eventual consistency, but they require intrusive rewrites of existing applications, making them impractical for most legacy systems.

There is a pressing need for a non-intrusive approach that enables local operations in geo-distributed applications without requiring significant changes to the application code. Our proposed approach addresses this challenge by externalizing consistency management from the application business logic. By separating these concerns, applications can maintain consistent behavior across geo-distributed environments without forcing developers to rewrite or modify their core logic, thereby reducing the complexity and overhead of altering existing code.

To enable this separation, it is important to examine the existing consistency approaches that could be leveraged. Most of the existing consistency methods can be classified into two approaches: state-based and operation-based:

- **State-based:** In state-based consistency approaches, each replica periodically exchanges and merges the state of its resource with others. The primary challenge here is that merging states often requires complex conflict resolution mechanisms, as each instance might have diverged. This approach is intrusive to the application since it involves direct manipulation and modification of the existing application state, which requires monitoring state changes and ensuring that these are accurately propagated across replicas.

- **Operation-based:** Operation-based consistency approaches, on the other hand, focus on transmitting the operations that cause state changes, rather than synchronizing the state itself. Each replica receives and applies operations, ensuring that all instances eventually reach the same state. A major advantage of this approach is that operations can be applied asynchronously and independently at each replica, which makes it more resilient to network partitions and conflicts are resolved at the operation level. This approach is less intrusive because it avoids directly manipulating the state of the resource and instead focuses on tracking the operations that lead to state changes.

Operation-based consistency approaches have the potential to address the pressing need for a non-intrusive solution that allow local operations in a geo-distributed application. Let us look at some of the existing operation based approaches and their issues.

For example, Operational Transformation (OT) [94], used in real-time collaborative editing application, ensures that even if users edit a document concurrently, the changes are merged correctly without loss of information. OT operates by transforming operations in such a way that conflicting edits are reconciled based on predefined rules, allowing the document or shared resource to remain consistent across all replicas. However, OT requires modifications to the application codebase to support its operation-handling mechanisms and specific data structures, making it less suitable for legacy systems or applications that cannot easily accommodate these changes.

Operation-based approaches, such as RAFT [15], provide strong consistency by coordinating operations across multiple replicas, ensuring that the same sequence of operations is consistently applied to all nodes, even in the presence of network partitions. However,

this coordination introduces overhead, as it requires frequent leader elections during a network partition, which can impact availability and cause delays.

On the other hand, operation-based CRDTs [95] aims to achieve strong eventual consistency [50] by allowing operations to be executed in any order across replicas without requiring real-time coordination. While this enhances availability by enabling local processing of operations, it comes at the cost of requiring modifications to the application code to work with specific data types, thus adding complexity to the implementation.

Although existing operation-based approaches ensure convergence, they face several challenges, particularly regarding performance during network partitions and their potential intrusiveness. Nevertheless, the operation-based concept remains promising, as it has the potential to manage consistency at the API level, external to the business logic of the application, aligning with the approach we envision. A key factor in understanding how operation-based consistency works is recognizing the distinction between stateless and stateful operations.

- Stateless Operations: These operations do not alter the state of the resource and typically involve read-only operations. For example, reading the current value of a counter or retrieving a string from a database are stateless operations.
- Stateful Operations: These operations modify the state of the resource and can lead to conflicts if not managed properly. For example, incrementing a counter by 10 ($increment_counter(10)$) is a stateful operation because it changes the current value of the counter.

Stateless operations can cause issues like stale read between replicated instances of a resource. In our approach, we are not considering these issues and focus only if an operation affects the state of a resource (i.e., stateful operations). Stateful operations inherently alter the state of resource within an application, and their impact on consistency and synchronization have higher significance.

By leveraging operation-based eventual consistency [96] and causal broadcast [97] techniques, we propose an approach to manage synchronization externally while ensuring a strong eventual consistency across geo-distributed instances.

Strong eventual consistency (SEC) [50] is a model that balances the trade-offs between strong [98] and eventual consistency. In SEC, replicas are allowed to process operations independently, ensuring high availability and low latency. Unlike eventual consistency, which only guarantees that all replicas will converge to the same state eventually, SEC

ensures that all operations are applied in a consistent order across replicas, resolving conflicts deterministically to achieve a consistent state.

The autonomy of replicas in this model can lead to concurrent operations occurring more frequently.A concurrent operation [99] occurs when two operations are executed simultaneously on different replicas without being aware of each others execution.

Our approach is centered on resolving these concurrent conflict issues while maintaining both a local-first and generic design. It ensures that consistency can be achieved without being intrusive to the application.

We detail our approach by categorizing stateful operations based on their commutative and idempotent properties. These properties are essential to identify how the replicas can converge to the same value during concurrent operation conflict. By analyzing how these operations affect a resource and behave when applied multiple times (idempotency) and in different orders (commutative), we develop two further categorize for stateful operations.

8.2 Extending stateful operations

We further categorize stateful operations into two:

- ***Replace operations:*** operations where the new state of the resource is independent from its existing value.
- ***Iterative operations:*** operations where the new state of the resource is dependent on its existing value.

We further explore these two categories by analyzing their commutative and idempotent properties, as they are key in determining a conflict resolution strategies for concurrent operations, to ensure that values across replicated resources are consistent.

Replace operations are typically idempotent, meaning that applying the same operation multiple times produces the same result as applying it once. This idempotency property simplifies consistency management because repeated execution of the same operation does not lead to varying outcomes. This behavior is particularly useful while retrying an operation when there is no response from a remote site, like during a network partition.

For instance, setting a counter to a specific value ($set_counter(50)$) or overwriting a string with new content ($set_string(Hello, World!)$) are examples of Replace operation. Regardless of how many times these operations are applied, the counter will always be set

to 50, and the string will always be updated to $Hello, World!$. This predictability makes Replace operations easier to manage, as they can be safely retried in the event of network failures or partitions without introducing inconsistencies.

Meanwhile, *Replace operations* are non-commutative, meaning that the order in which these operations are applied can significantly affect the final state of the resource.

For example, consider a counter that is first set to 100 and then replaced with 50 ($set_counter(100)$ followed by $set_counter(50)$), it will result in the value being 50. If these operations are applied in reverse order, i.e., setting the counter first to 50 and then to 100, the final value will be 100, i.e., the values will differ depending on the sequence of operation.

Similarly, with a string, if the string is first set to $Hello$ and then replaced with $World$ ($set_string(Hello)$ followed by $set_string(World)$), the final result will be $World$. However, if the operation order is reversed, i.e., setting the string first to $World$ and then to $Hello$, the final result will be $Hello$. This non-commutative property highlights the importance of maintaining the same order of operation across all replicated instances, to ensure a consistent state for the resource.

Unlike Replace operations, *Iterative* are not idempotent because applying the same operation multiple times leads to different results.

For example, incrementing a counter by a specific value ($increment_counter(10)$) is an Iterative operation. Each execution of this operation, increases the counter value by 10, demonstrating that iterative operations result in cumulative changes.

Similarly, appending a character to a string ($append_string(!)$) adds it to the end of the string, with each execution it adds a new ! character to the string. Due to their non-idempotent nature, Iterative operations require careful management to ensure that an operation is not executed more than once, thereby maintaining a consistent resource across different instances.

Iterative operations, which involve incremental changes based on the current state of the resource, can be either commutative or non-commutative, depending on the nature of the operation.

Iterative operations that are commutative, are those where the order of execution does not affect the final outcome. For example, consider a counter where two operations, $increment_counter(5)$ and $increment_counter(10)$, are applied. The final result will be the same, regardless of the order the increments are applied; the counter will reflect a cumulative increase of 15.

On the other hand, non-commutative iterative operations are sensitive to the order. For instance, consider two concurrent operations: *append_string* (abc) and *append_string* (def) on a replicated string resource. If abc is appended first and then def, the final string will be $abcdef$. However, if the order is reversed, appending def first and then abc, the final string will be $defabc$. In this case, the order of operations matters, making the operation non-commutative.

Both of these scenarios belong to *Iterative operations*, as they depend on the existing value of the resource to create a new one. The distinction between commutative and non-commutative iterative operations is crucial towards understanding if maintaining a consistent order of execution across replicas is essential to achieving a consistent geo-distributed resource.

Our approach relies on a Reliable Causal Broadcast (RCB) [97] messaging system to ensure that each operation is propagated to the relevant remote sites. RCB offers two key advantages: first, it inherently follows a causal delivery [100] system for messages, ordering them according to logical clocks as described by Lamport [101]; second, it guarantees that each message is delivered exactly once to each replica instance, ensuring reliable and consistent communication across geo-distributed environments.

This guarantee is not sufficient, as two concurrent operations can still occur between two replicated instances. While RCB can detect the concurrency, by checking if the operations are causally related, it does not resolve the conflicts that arise from such situations. This leaves the resource in a state where concurrent updates may result in inconsistencies across replicas, highlighting the need for additional mechanisms to ensure proper conflict resolution.

Our contribution involves creating an approach to resolve conflicts when two operations are concurrent. There can be multiple combinations of operations belonging to *Iterative* or *Replace* and *commutative* or *non-commutative*.

For example, if operation (1) is *increment_counter*(5) and operation (2) is *decrement _counter*(10), the resulting combination will be *Iterative* and *commutative*.

If operation (1) is *increment_counter*(5) and operation (2) is *set_counter*(8), as the first operation is *Iterative* and another is *Replace*, the resulting combination will be *Replace* (explained later in Section 8.1) and *non-commutative*, as replace is always *non-commutative*.

If operation (1) is *append_string*("abc") and operation (2) is *append_string*("def"), the resulting combination will be *Iterative* and *non-commutative*.

We create a classification model with these three combinations of concurrent operations for any resource to ensure that any replicated instance can be converged to a single value with strong eventual consistency guarantees.

8.3 Classifying Operations

With *Replace* and *Iterative operations*, as described in Section 8.2, we generalize three classes to resolve concurrent operations. Each class has a dedicated strategy that resolves two concurrent operations.

We are addressing operations and resolving concurrent conflicts external to the application, without requiring consensus mechanisms. These operations can be applied locally to a site, even during a network partition, allowing for continued functionality in isolated environments. Our approach is generic and can be applied to any application, ensuring flexibility and robustness without necessitating changes to the core application logic.

This approach requires users or developers to understand which combination a set of concurrent operations belongs to. We introduce a method to express classes for combinations of operations, which will be discussed later in Section 8.4.

In this section, we focus on defining these classes, as illustrated in Table 8.1. Let's now delve into each of these classes in detail.

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 8.1: Class representation of concurrent operations and their conflict resolution strategies.

8.3.1 Class 1: Iterative & commutative operations

The resulting combination of two iterative operations can create in it being commutative (i.e.,the order of operations does not matter at each replica). In such a combination, Cheops need to ensure that an operation is applied exactly once. An RCB like message delivery can ensure that an operation is applied exactly once per replica.

For example, consider a replicated Positive-Negative (P-N) counter across two sites: *Site* 1 and *Site* 2. Two operations, *increment_counter*(10) from *Site* 1 and *decrement_counter*(5) from *Site* 2, are applied locally and concurrently at each site, later, they are propagated to remote replicas. Upon receiving at the remote sites, Cheops detects that they are concurrent, with the help of RCB like protocol.

If the operations belonging to this class is applied multiple times on a replica, it can potentially cause the replicas to diverge. This is because iterative operations are inherently non-idempotent, meaning repeated application can produce different results. However, these operations can still be applied in any order (e.g., applying *increment_counter*(10) after *decrement_counter*(5) or vice-versa) and the values at each replica will remain consistent, ensuring convergence despite the order of application. This is due to the commutative nature of iterative operations, which allows them to be reordered without affecting the final result.

The only requirement for this class is the exact once delivery of operations, which is already guaranteed by RCB. As a result, we can ensure that concurrent operations belonging to this combination will converge, since the commutative nature of these operations combined with reliable delivery ensures consistency across replicas.

8.3.2 Class 2: Replace & non-commutative operations

This class of concurrent operations includes at least one *Replace* operation, leading to a combination of *Replace* & $non - commutative$ operations. A combination of a *Replace* operation with any other, such as an *Iterative* or another *Replace*, will always result in $non - commutative$ behavior. In other words, each replica must apply the operations in the same order to ensure convergence.

The possible concurrent operation combinations are:

- **Replace & Replace:** For example, two *set* operations on a replicated string resource occur concurrently from different sites, such as *set_string*(”*foo*”) and *set_string* (”*bar*”). Since these operations are $non - commutative$, the final value

at each replica will depend on the order in which the operations are applied. To ensure consistency, replicas must apply these operations in the same order across all sites.

- **Replace & Iterative:** For example, a *set* and an *append* operation on a replicated string resource are concurrent, such as $set_string("Hello")$ and $append_string$ $("World!")$, these operations are $non - commutative$. To ensure consistency, the order of application must be the same across all replicas.

Both combination of operations are non-commutative, as the existing value is at least replaced once with a new one (due to the *replace* operation). Hence, we need a deterministic order to resolve the conflict.

For the first combination, *Replace* & *Replace*, in our local-first approach, these operations are applied locally at each site first and then propagated to the remote replicas. Upon receiving the remote operations, RCB identifies the concurrency by analyzing the causal order, recognizing that the two operations occurred simultaneously and need to be resolved to ensure consistency across replicas.

Once the concurrency is identified, a deterministic order based resolution, like Last-Writer-Wins (LWW) is applied at each site to resolve this conflict. In our scenario, the operation $set_string(foo)$ is applied as the LWW at both sites, ensuring that the values eventually converge.

For the second combination, consider a messaging application where two users concurrently update the same replicated key from different sites, *greetings*, which initially holds the value $Hello!$. User A performs a *replace* operation, setting the value to $Good\ Morning!$, from the first site, while user B performs a concurrent *iterative* operation, appending $How\ are\ you?$ to the original message, resulting in $Hello!\ How\ are\ you?$, from the second site. In a LWW approach, only the most recent update would be retained, such as $Good\ Morning!$, with update from user B being discarded to ensure consistency across all replicas, as shown in Figure 8.1.

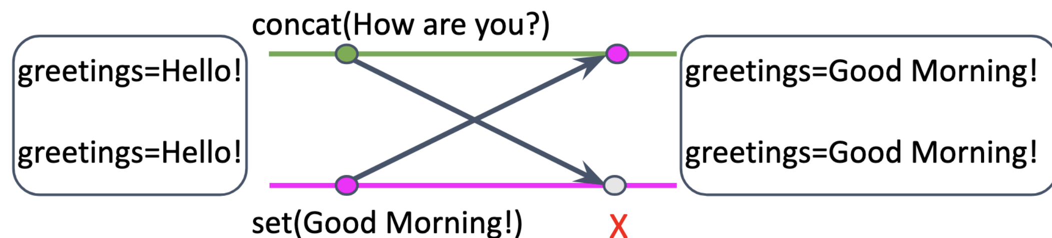

Figure 8.1: Replace & Iterative: solution 1

LWW can result in some concurrent operations not being applied at each site, but it ensures strong eventual consistency between instances, which may be sufficient for certain applications, as illustrated in Figure 8.1. However, there are scenarios where an application cannot afford to lose any of the concurrent operations, particularly when an *Iterative* operation is involved, such as the *append* in a messaging application. In these cases, preserving all operations is crucial, as discarding *iterative* updates, like appending new content to a message, could lead to missing important data, LWW alone is insufficient for such use cases.

To address this challenge, we propose to further enhance the LWW approach based on our model, as illustrated in Figure 8.2. Operations from user A and B are first applied locally at their respective sites and are then propagated to remote sites. Upon receiving these operations, RCB detects the concurrency, but both operations are still applied at the remote sites to ensure that no concurrent updates are lost.

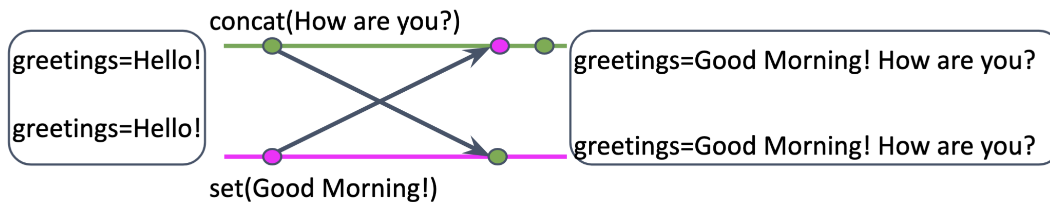

Figure 8.2: Replace & Iterative: solution 2

Now the value at one site will be $Good\ Morning!\ How\ are\ you?$ and other will be $Good\ Morning!$, to converge these two to the same value, we apply the append operation again onto the site where $Good\ Morning!$ operation was last applied, to ensure that we reach a common value $Good\ Morning!\ How\ are\ you?$ across all sites. In other words, we deterministically re-apply an *iterative* operation (such as an append) onto a *replace* operation (such as a set), where the *replace* is the last applied operation. This re-application occurs after both operations have been initially applied to all sites. This ensures that the values converge across all sites, preserving both updates and achieving consistency.

8.3.3 Class 3: Iterative & non-commutative operations

This class combines two different types of concurrent *iterative* operations that result in a *non-commutative* combination. For example, a *Append* and $leftshift$ operation for a string resource that are concurrent or two Append operations. Let us look at it in detail.

Consider a string $c = "foo"$ with two concurrent operations, *Append*("*bar*") and $leftshift(2)$, initiated from different replicas. Since we follow a local-first approach, the

value of the resource at each replica will diverge after applying the operations. At *site* 1, the *Append*(*"bar"*) operation results in $c = "foobar"$, while at *site* 2, the $leftshift(2)$ operation results in $c = "ofo"$. Even if we apply standard conflict resolution methods, such as last-writer-wins, the value at each site will remain divergent. A similar issue arises with two concurrent Append operations, where convergence between replicas cannot be guaranteed without a more specific approach to conflict resolution.

Another example is a combination of addition(+) and multiplication (X) operations. For a replicated integer $c = 10$, at *site* 1, the user applies *add*(5), resulting in the resource being $c = 15$, concurrently at *site* 2, another user applies *multiply*(2), resulting in the resource being $c = 20$. If we try to merge them with last writer wins or any operation-only based solutions (not changing the states explicitly), the state at each site will still remain diverged.

There are specific solutions [102, 103, 104, 105] to solve these issues but they are either intrusive to the existing application logic as they require to introduce additional data type (like CRDT) or code, or they are based on consensus, which violates the local-first principle, to ensure convergence of a replicated resource value.

To avoid being intrusive, we re-utilize an approach which we previously introduced in Cheops [17] called *Resource_logic*, that allow the execution of a dedicated (pre-defined) code while applying an operation to the resource. It is situated external to the application (at Cheops level) at each site and can control operations explicitly.

Since we are dealing with consistency, we call this approach as *Consistency_logic* The input to *Consistency_logic* can vary based on the each algorithm and the output will be a series of operations performed to ensure that all the replicas are converged.

Consistency_logic is applied locally at each site after RCB detects the conflict. *Consistency_logic* is an interface, written in go-lang [106], where the user can add any code that can guarantee an eventual convergence of all replicas.

We present an approach that can serve as the default solution for this class of concurrent operations. This approach, developed based on the concept introduced with Peritext [103], addresses conflicts by ensuring a consistent resolution of operations. It offers a method for handling divergence in concurrent operations, like the *append* and $leftshift$ examples, ensuring that replicas converge to a unified state even in the presence of such conflicting operations.

Even though Peritext is designed for collaborative rich text editing, we can generalize it for any operation. We externalize Peritext algorithm, to better adapt to Cheops

orchestrator to ensure consistency in this class of operations.

Peritext for Class 3: Iterative & non-commutative operations

Peritext [103] is designed using the Conflict-free Replicated Data Type (CRDT) approach for collaborative rich-text editing. It leverages the inherent properties of CRDTs, where operations are commutative and idempotent by default, ensuring that changes made by different users can be applied in any order without conflicts.

CRDTs are intrusive as they require changes to the underlying data structures and application logic. However, Peritext, can be constructed outside of it, allowing it to be adopted to Cheops. Peritext addresses concurrency issues by implementing a specific ordering mechanism for handling concurrent operations. Here's how it generally works:

1. Operation Metadata: Each operation in Peritext is tagged with metadata that includes a unique identifier (often based on a combination of the site ID and a logical clock [101]). This metadata helps in determining the order of operations.

2. Commutativity and Associativity: The operations in CRDTs are designed to be commutative and associative. This means that the order in which operations are applied does not affect the final result, ensuring that all replicas converge to the same value.

3. Handling Concurrent Operations: When two operations are concurrent (e.g., two Append operations are performed concurrently), Peritext will apply them in a consistent order based on their metadata. This ensures that both replicas will end up with the same result.

One of the key mechanisms that Peritext employs to manage concurrency is with metadata and the use of unique identifiers (IDs) for each operation. In our Cheops adopted version of peritext, Cheops assigns metadata from a local site with each operation and this is given to the algorithm, as illustrated in Algorithm 1.

The metadata in Peritext plays a pivotal role in resolving conflicts between concurrent operations. Each operation ID typically consists of two parts: the site ID and the logical timestamps.

- **Site ID**: This ensures that each operation can be traced back to the specific user or site that generated it.

- **Logical Timestamp**: This provides a way to order operations, even when they are generated concurrently. Logical timestamps might be simple counters that increment with each new operation like the lamports Clock [101], or they might involve more complex structures like version vectors that track the causal relationships between operations. Our approach by default use an RCB, for logical timestamps, to Identify the causal order.

Algorithm 1 Consistency Logic for Concurrent Operations with Cheops and Peritext

Require: State of the local resource before applying any concurrent operations: *state*
Require: List of concurrent operations to be applied with metadata: *operations*
Ensure: Final state after all operations are applied: $finalState$

```
 1: function ApplyOperation(state, operation)    ▷ Applies a single operation to the
    current state.
    return operation.execute(state)
 2: end function
 3: function ResolveConflict(op1, op2) ▷ Resolves conflicts between two operations.
 4:     if op1.metadata.id < op2.metadata.id then return {op1, op2}
 5:     elsereturn {op2, op1}
 6:     end if
 7: end function
 8: function ProcessConcurrentOperations(state, operations)
 9:     orderedOps ← [ ]                        ▷ Empty list to store ordered operations
10:     while operations is not empty do
11:         op1 ← first operation from operations
12:         op2 ← second operation from operations
13:         resolvedOrder ← ResolveConflict(op1, op2)
14:         Append resolvedOrder to orderedOps
15:         Remove op1 and op2 from operations
16:     end while
17:     Rollback current_state to state
18:     for all op in orderedOps do
19:         state ← ApplyOperation(state, op)
20:     end for
        return state
21: end function
```

Let us look at an example for two concurrent *append* operations in a replicated string.

Consider a scenario, with two replicas of a string, both initially containing the value foo. For *Class* 3 operations, Cheops stores the *state* of a resource before applying an

operation, this is necessary for the algorithm. Cheops identifies that an operation belongs to *Class* 3.

At *Replica* 1, a concatenation operation appends the string *append_string*(*bar*), resulting in the local value being $foobar$. Concurrently, at *Replica* 2, a concatenation operation *append_string*(*biz*), resulting in the local value being $foobiz$. Once an operation is received by Cheops, it assigns a unique ID based on the site of the operation, as discussed with Cross collaboration from Section 7.2.1. Since Cheops follows a local-first approach, the operations are applied locally first before being propagated to the remote replica to ensure consistency across both instances.

Once these local operations are applied, Cheops need to synchronize them with the other. This means that *Replica* 1 receives the operation from *Replica* 2 (*append_string* ("*biz*")), and *Replica* 2 receives the operation from *Replica* 1 (*append_string* ("*bar*")). Now, both replicas have a list of operations, with each needing to ensure that all operations are applied in the same order (since the operations are non-commutative), so that both replicas can converge to the same final value.

Cheops identifies that they are concurrent, since RCB is not able to establish a happened-before relation between the two operations, implying that they are concurrent. Cheops uses the *ProcessConcurrentOperations* function, described in Algorithm 1, to handle the concurrent operations in a way that ensures consistency.

The key part of this process is the conflict resolution mechanism. Since the two operations were applied concurrently, the algorithm uses metadata attached to each operation from Cheops, such the as unique operation IDs, to resolve any conflicts in operation order. The algorithm compares the unique IDs of each operation. In this case, the operation from *Replica* 1 has a smaller ID than the one from *Replica* 2, so it is applied first. As a result, the operation *append_string*("*bar*") has a precedence over *append_string*("*biz*").

Once an order is obtained, Cheops at *Replica* 2 instance rollback the resource to a previous value before the concurrent operation was applied. The operations are applied to the original string foo in the local site in the new order. First, *append_string*("*bar*") is applied to foo, resulting in the intermediate state $foobar$. Then, *append_string*("*biz*") is applied to this intermediate state, resulting in the final state $foobarbiz$. The process is performed locally at each replicated site, by their local Cheops. All replicas follow the same procedure, ensuring that they apply the operations in the same order. As a result, both replicas eventually converge to the same final state of $foobarbiz$.

We have illustrated our approach to convert an algorithm (Peritext) with *Consistency*

logic to a non-intrusive solution, compared to its current CRDT based approach. We adopt this Peritext based Cheops consistency approach as a default approach towards solving *Class* 3 concurrent operations.

Based on the user requirement, *Consistency logic* can further include any approach such as [102, 104, 105], to be adapted to Cheops, making them a non-intrusive solution to ensure consistency.

8.4 How can Cheops map a class to an operation?

Our approach involves developing a classification model for every possible combination of concurrent operations. Each class represents a specific resolution strategy to handle these concurrent operations. The operations are unique to each resource and may extend beyond basic CRUD. How can Cheops generically map any combination of concurrent operations for a resource to a corresponding class?

We propose an approach that involves creating a matrix, referred to as the *Consistency matrix*, which allows Cheops to determine the appropriate class for any combination of possible concurrent operations. The *Consistency matrix* is constructed for each resource data type, with operations listed along both the rows and columns, forming a square matrix.

This matrix-based approach offers flexibility, allowing it to be applied generically to any resource while simplifying Cheops interpretation. The input to the matrix comes from the user, who provides the appropriate class mappings for each pair of operation combinations on a given resource.

For example, Table 8.2 portrays a *Consistency matrix* for a P-N counter resource. The matrix outlines a solution for each possible concurrent operation combination for a P-N counter assigned to a class. Each of these class will have a specific semantics to resolve any concurrent conflict issues that can arise, as described in Section 8.3, for a replicated P-N counter. If the concurrency is identified by the RCB, Cheops uses this matrix is to resolve conflicts. Table 8.2 portrays interactions for three operations: *increment*, *decrement* (increments or decrements the value), and *set* (sets a value).

From Table 8.2, if an *increment* and *decrement* operations are concurrent, they follow *class* 1 semantics, i.e., applying each operation once per replica, as described in Section 8.3. If an *increment* or *decrement* operation is concurrent with a *set*, they follow *class* 2 semantics, as described in Section 8.3. If two *set* operations are concurrent, they

Operations	Increment	Decrement	Set
Increment	Class 1	Class 1	Class 2
Decrement	Class 1	Class 1	Class 2
Set	Class 2	Class 2	Class 2

Table 8.2: Operation matrix for P-N Counter

again follow *class* 2 semantics. This *Consistency matrix* allows Cheops to take necessary resolution strategies to ensure that all the replicas are converged. Depending on each resource, the matrix changes.

Let us consider another string resource. Table 8.3 portrays a string data type with five operations: *LeftShift* & *RightShift* (rotate the sequence of characters within a string by a certain value), *Set* & *Delete* (set a string or delete the string value) and *Append* (Append a value to the existing one).

Operations	Left Shift	Right Shift	Set	Delete	Append
Left Shift	Class 1	Class 1	Class 2	Class 2	Class 3
Right Shift	Class 1	Class 1	Class 2	Class 2	Class 3
Set	Class 2	Class 2	Class 2	Class 2	Class 2 or 3
Delete	Class 2	Class 2	Class 2	Class 2	Class 2 or 3
Append	Class 3	Class 3	Class 2 or 3	Class 2 or 3	Class 3

Table 8.3: Operation matrix for a String

The Table 8.3 portrays, if any combination of *LeftShift* and *RightShift* operations are concurrent, semantics from *Class* 1 can resolve it. If *Set* or *Delete* is concurrent with *LeftShift* or *RightShift*, then *Class* 2 semantics can be applied to resolve the conflict. If *Append* is concurrent with *LeftShift* or *RightShift*, then *Class* 3 semantics, as described in Section 8.3, can be applied to resolve the conflict i.e., with *consistency logic*. Similarly, for the remaining operations.

Let us look into a Pod resource in Kubernetes application, as illustrated in Table 8.4. We consider three operations, *Apply* (applies a new configuration to the resource), *Patch*

(a partial update to the resource, where only specific parts of the resource are modified without affecting the entire resource) and *Replace* (the resource is replaced entirely, often meaning the existing resource is deleted and a new one with the provided configuration is created).

Operations	Apply	Patch	Replace
Apply	Class 2	Class 2	Class 2
Patch	Class 2	Class 2	Class 2
Replace	Class 2	Class 2	Class 2

Table 8.4: Operations matrix for a Pod resource

Since, the Kubernetes configuration consist of Key-Value pair data, there is no *iterative* operations involved. This implies the three operation combinations belong to the same *Class* 2 semantics. Hence, if there is a concurrent operation combination among these three, a deterministic approach such as Last-writer-wins can be applied to resolve the conflict.

Operations	Update	Rebuild	Resize
Update	Class 2	Class 2	Class 2
Rebuild	Class 2	Class 2	Class 3
Resize	Class 2	Class 3	Class 2

Table 8.5: Consistency matrix for OpenStack operations on a VM

Let us look at another resource, a Positive-only Counter (P-Counter). For a P-Counter, the value cannot go below 0. If the value falls below 0, the application should return an error for the operation, as it violates the definition of the P-Counter. To deal with concurrent operations, the matrix is similar to a P-N Counter as shown in Table 8.2. For such a resource, if the value falls below 0, the matrix cannot be applicable anymore, as this is an inherent condition for the P-Counter, that is internal to the resource structure and Cheops does not try to capture this due to the non-intrusive principle. We discuss how Cheops can guarantee the consistency between such operations, like in the replicated P-Counter scenario, in the next paragraph.

8.5 Handling exceptions if eventual convergence fails

For a scenario involving a P-Counter, Cheops alone cannot address the issue using only a consistency matrix. To illustrate this with a concrete example, consider a replicated P-Counter with an initial value of 100, and two concurrent operations: *decrement*(50) from *replica 1* and *decrement*(51) from *replica 2*. Each operation will succeed locally at its respective site, but when propagated to remote replicas, these operations will cause an error as the counter value falls below zero.

The consistency matrix is designed to identify and resolve classes of concurrent operations. However, the issue with a P-Counter extends beyond typical concurrency problems, as it involves violations rooted in the internal structure of the resource itself. In cases like these, where the resource integrity is breached, an error should be reported automatically by the application instance.

Cheops, by default, collects these error responses from each site through the local agent in a synchronous manner to assess the status of operations. This feature, known as *partial error*, was introduced in previous work [17] and further elaborated in Section 7.2.2. By analyzing the list of responses from each site, users can determine whether an operation was successful at any particular instance.

We extend the concept of *partial error* to also detect conflicts that arise not only from concurrency issues but also from violations of resource-specific properties. For instance, in the case of the P-Counter, when its value drops below zero, the application generates an error that Cheops captures via the *partial error* mechanism. Cheops can interpret this error, enabling the implementation of a deterministic approach to resolve these specific issues.

Cheops adopts a two-fold strategy to handle such cases. First, it identifies whether the error is due to a violation of resource-specific values, such as a P-Counter falling below zero. The second step is to devise a method to resolve the detected error.

To address this, we propose a new mechanism based on *partial error*, termed *Cheops hooks*, inspired by the concept of Git hooks [107]. Git hooks are scripts that automatically execute when certain events occur in a Git repository. Similarly, in our approach, an event is triggered when a Cheops agent detects an error in the local instance of an application. This enables automated responses based on the specific error.

Cheops remains non-intrusive to the application and requires user-provided input to identify such errors. We introduce an interface, similar to the existing *Consistency logic*,

called *Hooks logic*, where the user can define a function that allows Cheops to identify the necessary error from the list of *partial error*. The interface also provides another function for resolving these errors.

Code 8.1: Hooks Logic definition for P-Counter

```
// Hooks_logic:

// Pseudocode for detecting and resolving P-Counter violations
with rollback and operation decomposition

Detection(partial_errors E):
    FOR each error in E:
        IF error type is 'P-Counter underflow':
            Identify the specific site and instance where the violation occurred
            Log the violation details for further resolution
            RETURN violation detected with error details
        ENDIF
    ENDFOR
    RETURN no violation

Resolution(partial_errors E, operations Op, previous_state S):
    // Rollback to the previous consistent state

    IF error type is 'P-Counter underflow':
        Restore the P-Counter to the previous state (S)
        before the conflicting operations were applied.

    // Decompose operations and apply decrement(1) sequentially


    IF last operation in Op is decrement:
        decrement_value = operation.value //for decrement(50), 50 is assigned here

        // Decompose the decrement into decrement(1)

        FOR i = 1 to decrement_value:
            IF counter_value > 0:
                counter_value -= 1  // Apply decrement(1)
            ELSE:
                // If the counter has reached zero, stop applying further decrements
                BREAK
            ENDIF
        ENDFOR

        // Log if operation could not be fully applied due to reaching zero
        IF i < decrement_value:
            Log "Partial application: operation stopped at counter value 0."
        ENDIF
    ENDIF
    RETURN result back to the user
```

Once Cheops identifies the error, it relies on the *hooks logic* to apply the resolution defined by the user. A deterministic solution can be implemented in the *hooks logic*. One possible resolution could be to lock the entire replicated instance, preventing any further operations, and notify the user to take manual corrective actions.

Another possible deterministic solution is the concept of operation decomposition. Op-

eration decomposition involves breaking down an operation into smaller sub-operations, which, when performed, collectively fulfill the original operation.

For example, consider the P-Counter scenario with an initial value of 100 and two operations: *decrement*(50) from *replica 1* and *decrement*(51) from *replica 2*. Using operation decomposition, the *decrement*(50) can be divided into 50 individual *decrement*(1) operations. A *Hooks logic* for a P-Counter is defined in Code 8.1. It consist of two functions *Detection* and *Resolution*.

Detection is responsible for identifying errors from the list of partial errors that cause the P-Counter to drop below zero. This function inspects each partial error, pinpoints the specific site and instance where the violation occurred, and logs the violation for further action.

Resolution handles the correction of the detected violation by rolling back to the previous consistent state with each local Cheops agent. It then decomposes the conflicting operations into smaller increments of *decrement*(1), applying each decrement sequentially until either the counter reaches zero or the operation completes. If the counter reaches zero before completing the operation, it logs the *partial error*, ensuring the system maintains strong eventual consistency.

The approach with *Hooks* ensures that Cheops can guarantee a strong eventual consistency even during conflicts beyond concurrent operations.

Limitations

As far as we know, this is the first approach to externalize consistency for geo-distributed applications in a generic and non-intrusive manner, while supporting local-first operations by default. At this stage, our model represents an initial step towards our goal and has certain limitations, which pose open questions for the research community. We outline these limitations below:

Restricted API access

Our approach depends on the application API being accessible to Cheops; however, this is not always the case. We provide a two examples to illustrate this challenge:

- For two *append* operations on a replicated string resource, if only these operations are exposed through the application API, our proposed approach using peritext, as

described in Section 8.3.3, may not be effective. This is because Cheops need an API to read and overwrite the string. The challenge arises from relying on the resource native data types, rather than using a custom data type like CRDT, which limits the available operations.

Strong Eventual Consistency vs. Correctness

Our approach ensures that values across replicas eventually converge; however, we do not guarantee that the converged value is the correct one. This limitation can lead to issues, particularly in scenarios where correctness is crucial, such as in financial systems.

For example, consider a banking application where two users attempt to withdraw 100 concurrently from different locations but from the same account, which only has a balance of 100. With Cheops, both withdrawals might succeed locally at each site since the operations are processed independently. Eventually, the system will converge to a state where both withdrawals have been applied, but the final balance will show a negative value (e.g., -100), which violates the rule that a bank account balance cannot drop below zero.

This is unacceptable because the invariant that the account balance must remain above 0 is not upheld. Our approach cannot handle scenarios that require adherence to such strict rules, as it is designed to be generic rather than tailored to specific use cases.

While we guarantee strong eventual consistency, meaning the system will converge to the same value across instances, we do not guarantee that the converged value will always be valid from the application business logic.

8.6 Validation

In this section, we validate the proposed consistency approach for Kubernetes application. The validation demonstrates how our approach ensure a strong eventual consistency across geo-distributed independent Kubernetes clusters for various resources, including Pods (Basic units of Kubernetes applications) and Deployments (Manages replicated sets of pods).

Each Kubernetes resource is assigned a consistency matrix, which classifies potential concurrent operations and provides the appropriate conflict resolution strategy. The experimental setup is the same as described in Section 6.3.

8.6.1 Kubernetes Pods

A pod represents the smallest deployable unit in Kubernetes, consisting of one or more containers. In this validation, we tested concurrent operations such as apply, replace, and patch on the same pod across different sites. The Consistency Matrix for a Pod was already illustrated earlier in Table 8.4.

The consistency matrix for a Kubernetes pod, as shown in Table 8.4, covers three operations: Apply, Patch, and Replace. These operations are classified based on whether they involve changes to the entire pod configuration or specific aspects of the pod, such as environment variables or container images.

We create a pod foo, replicated across two sites, *Site* 1 and *Site* 2, created by the Cheops CLI from *Site* 1, `cheops --cmd kubectl create pod foo -f pod.yaml --scope {Site 1 & Site 2}`. All the operation requests on either of these sites, are applied locally and propagated to the other site.

We applied the following concurrent operations:

- First operation is applied at *Site* 1: apply operation to change the container image from nginx:v1 to nginx:v2. The operation is performed with Cheops CLI as, `cheops --cmd kubectl apply pod foo -f pod1.yaml`.
- Second operation is applied at *Site* 2: patch operation to change the pod environment variable values. The operation is performed with Cheops CLI as, `cheops --cmd kubectl patch pod foo -f pod2.yaml`.

Once both operations are propagated, Cheops identifies that they are concurrent by the RCB (as it fails to create a causal relation between operations). This indicates Cheops to refer the Consistency matrix provided for the Kubernetes pod resource, as illustrated in Table 8.4.

The resolution for the combination of *Update* and *Patch* concurrent operations, is based on *Class* 2 (Replace & any) semantics, therefore, Cheops need to apply a last-write-wins strategy to resolve the conflict.

In our scenario, the operation from *Site* 1 gets a precedence as the last operation. At *Site* 1, the local operation *Update* is already applied, then, the remote operation *Patch* is received from *Site* 2, but *Patch* is not applied as *Update* wins. At *Site* 2, the local operation *Patch* is already applied, then, the remote operation *Update* is received from *Site* 1, *Update* is applied after the *Patch* operation on resource foo. This ensured that the pod foo, is converged at both sites after the concurrent operations.

8.6.2 Kubernetes Deployments

A deployment resource manages a replicated set of pods, ensuring that a specific number of replicas are running at any given time. In this validation, we tested concurrent operations such as scaling (resize), updating pod templates (update), and replacing the entire deployment configuration (replace).

The consistency matrix for Kubernetes deployments is illustrated in Table 8.6. This matrix covers three operations: Resize (adjusts the number of replicas), Update (changes the pod template), and Replace (replaces the entire deployment configuration).

Operations	Resize	Update	Replace
Resize	Class 2	Class 2	Class 2
Update	Class 2	Class 2	Class 2
Replace	Class 2	Class 2	Class 2

Table 8.6: Consistency matrix for Kubernetes Deployments

We create a deployment *bar*, replicated across two sites, *Site* 1 and *Site* 2, created by the Cheops CLI from *Site* 1, `cheops --cmd kubectl create deployment foo -f dep.yaml --scope {Site 1 & Site 2}`. All the operation requests on either of these sites, are applied locally and propagated to the other site.

We applied the following concurrent operations:

- First operation is applied at *Site* 1: Scale the number of replicas in the deployment from 3 to 5 (Resize). The operation is performed with Cheops CLI as, `cheops --cmd kubectl scale deployment bar --replicas=5`.
- Second operation is applied at *Site* 2: A deployment update to modify the pod container resources (CPU/memory limits). The operation is performed with Cheops CLI as, `cheops --cmd kubectl apply deployment bar -f dep1.yaml`.

As mentioned in the pod case, RCB within Cheops identifies that the operations are concurrent. Cheops then refers to the Consistency matrix for the Kubernetes deployment resource, as shown in Table 8.6, to determine the appropriate resolution method needed to ensure consistency.

The concurrent combination of *Resize* and *Update* operations, is classified as *Class* 2 (Replace & any), therefore, Cheops need to apply a last-write-wins strategy to resolve the conflict.

In our scenario, the operation from *Site* 2 gets a precedence as the last operation. At *Site* 2, the local operation *Update* is already applied, then, the remote operation *Resize* is received from *Site* 1, but *Resize* is not applied as *Update* wins as the last operation At *Site* 1, the local operation *Resize* is already applied, then, the remote operation *Update* is received from *Site* 2, *Update* is applied after the *Resize* operation on resource *bar*. This ensure that the deployment *bar*, is converged at both sites after the concurrent operation.

The result showed that Cheops correctly identified the concurrent operations using RCB and applied the necessary resolution strategy using the classes. This ensured that the Kubernetes deployment and pod converged to a consistent state across both sites. This validation confirms the effectiveness of our external consistency model in handling concurrent operations on geo-distributed Kubernetes application resources.

8.7 Summary

In this chapter, we introduced our approach to externalize consistency mechanisms for geo-distributed applications, focusing on maintaining synchronization across replicated resources in a non-intrusive and generic manner. Traditional methods, which often embed consistency mechanisms directly into the application logic, were found to be overly intrusive and unsuitable for legacy systems not designed for geo-distribution. To address these challenges, we proposed a methodology to externalize consistency management from the business logic of the application, allowing applications to maintain consistency across geo-distributed environments without requiring code modifications.

Our approach leverages operation based strong eventual consistency and causal broadcast techniques to manage synchronization externally. We introduced a classification to further classify stateful operations into replace and iterative categories. This classification allowed us to develop tailored conflict resolution strategies, ensuring that operations do not lead to diverged states across geo-distributed instances.

We also presented an extension of these categories into three classes based on their commutativity and idempotency properties, each with specific resolution strategies. Additionally, we provided a mechanism for Cheops to map combinations of concurrent operations to a class through a consistency matrix, enabling it to apply the required resolution

strategy for a resource, to guarantee consistency across sites.

Finally, we discussed how Cheops handles exceptions where strong eventual consistency might fail, particularly in cases where operations could create conflicts outside of concurrency such as a P-counter. By utilizing a hooks logic, Cheops can identify such conflicts and either halt further operations until resolved or apply a predefined resolution strategy. We demonstrated our approach on the Kubernetes container orchestrator application, with deployment and pod resources.

This chapter serves as a foundational step towards achieving non-intrusive, generic strong eventual consistency management for geo-distributed applications, while prioritizing a local-first approach. While our approach offers significant advantages, it also has limitations, particularly concerning restricted API access, which remain as an open area for further research.

CHAPTER 9

HANDLING DEPENDENCIES EXTERNALLY FOR GEO-DISTRIBUTED INSTANCES

This chapter focuses on my final contribution in this thesis. It addresses the third research question, described in Section 3.3. Cheops is a solution that relies on deploying a copy of an application at individual sites and creating an illusion of a single instance. A single instance implies, a request from one site should contain the same response as from another, not considering network partitions. This can be hard to achieve as described in Section 3.2, due to dependencies a resource might posses. In this chapter, I explain about these dependencies, caused by relationship between resources and how we can address them such that Cheops can maintain the illusion of a single geo-distributed application.

9.1 Managing Relationships

There are strong relationships between resources, like between a Pod and Secret in Kubernetes, as discussed in Section 3.2. Applications such as Kubernetes and OpenStack are designed to function as single instances, not geo-distributed. Cheops addresses this by combining individual instances to create the illusion of a single application. However, when geo-distributing resources, dependency issues arise since instances are not inherently connected. This can lead to one instance having a dependent resource while the other does not, as discussed in Section 3.2.

Another example in OpenStack is when a Virtual Machine (VM) is attached to a specific block storage volume for data persistence. This VM is initially deployed in *Site* 1, where both the VM and the volume are available. A user initiates a request with Cheops to replicate this VM with *Site* 2, where another instance of the Openstack application is running (managed by Cheops).

Cheops gets the request and creates a replica of the VM at *Site* 2, but it will eventually fail, since it relies on the volume. This volume is not available in *Site* 2 instance of Open-

stack. This contradicts the Cheops vision of having an illusion of a single geo-distributed application, derived from the single system image research [16]. If this was a true illusion of a single geo-distributed application, the VM replicated between these sites should have been successful.

Some approaches exist that model dependencies based on contexts, such as network dependencies [108], which focus on dynamically identifying dependencies between components in an ad hoc network. Other research includes finding structural dependencies based on the number of communication links [109] or identifying temporal changes within systems based on structural dependencies [110]. These approaches focus on managing dependencies within an application, they are not focused on seamless geo-distribution of resources.

One approach that stood out in our survey was Liqo [63]. In Liqo, resource reflection is a feature that allows for the seamless propagation of Kubernetes Pod dependent resources such as Services, ConfigMaps, Secrets, and Ingresses across multiple clusters. This ensures that the deployment in remote clusters have access to the necessary infrastructure.

This mechanism plays a key role in enabling multi-cluster workloads, ensuring that the same resources available in the local cluster are reflected in remote clusters. This approach automatically detects the dependencies from a local cluster and propagates them to the remote site, when a deployment is offloaded to another instance of Kubernetes.

Liqo has created an approach specific to Kubernetes, understanding the schematics of its configuration to identify if a dependency exist or not. Although we aim for a similar approach, Liqo is not applicable to all applications. To the best of my knowledge, a method to manage dependencies to facilitate geo-distribution of a resource does not exist that matches the cheops principles (generic and non-intrusive).

9.2 Relationship model

To create an illusion of a single geo-distributed application, Cheops need to identify and solve these relationships. To identify them, we created a classification, by analyzing applications such as Openstack, Kubernetes, etc. We already published an initial version of this relationship model [13].

We identified different relationships that occur between resources, such as pod and secret in Kubernetes, VM and block storage volume in Openstack and project and tex files in Sharelatex. We observed that these relations need to be reflected among different

instances of the same application while trying to geo-distribute these resources. These studies and observation, lead us to create a classification model with four different classes.

In this classification, we use the term "state" of a resource to indicate whether it is currently active or inactive. It is illustrated in Figure 9.1 and outlines four relations:

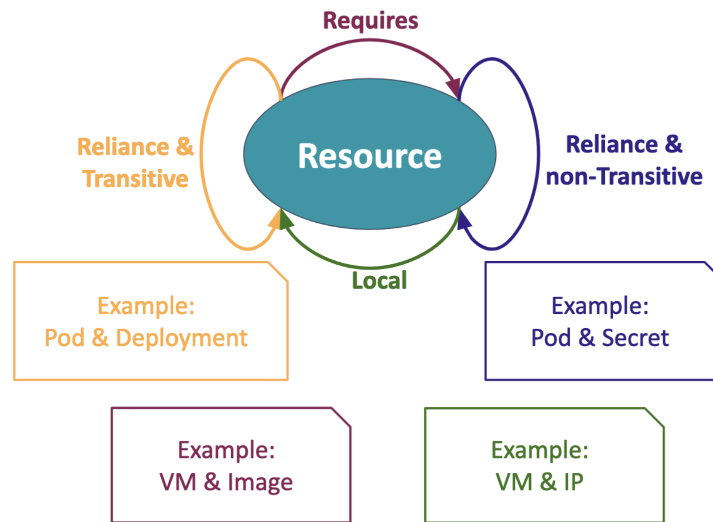

Figure 9.1: Cheops relationship model

- ***Requires***: Resource A requires B, portrays a temporal relation. This relation is largely observed in a request initiated by a user. Once a request is completed, it has no further purpose for existence.

- ***Reliance & non-Transitive***: Resource A relies on B, portrays a permanent relation, if the connection between them is broken, it will affect the state of either or both of them. Non-Transitive implies that both of the resources are independent and their lifecycle is not intertwined.

- ***Reliance & Transitive***: Resource A relies on B, portrays a permanent relation, if the connection between them is broken, it will affect the state of either or both of them. Transitive implies one resource controls the lifecycle of another including operations.

- ***Local***: Resource A is local to an environment (a specific value, generally hard to reproduce) or a location, if it is changed, it can impact the state of the resource.

In the later part, I explain these relationships in detail by breaking down each of these types of relationships. This includes their characteristics, how they interact within resources, and how we can manage them effectively while geo-distributing a resource.

9.2.1 Requires Relationship

The *Requirement* relationship defines a dependency between two resources that is necessary for the execution of a particular operation. This relationship is vital during the operation; if the communication between the resources is severed while the operation is in progress, it will terminate unsuccessfully. Restoring the communication can often resume the operation.

However, once the operation is successful, the communication between the resources has served its purpose and the dependency ends. Hence, this relation is temporal and does not impact the state of the resource.

For example, in an OpenStack application, creating a VM requires an Image. During the VM creation process, the VM service must communicate to another to fetch the Image. The communication between the VM (nova) and the Image (glance) services is critical for this operation; if any of the service becomes unavailable during the creation process, the operation will fail. However, once the VM is successfully created, the communication between nova and glance is no longer required. The dependency between the VM and the Image service becomes irrelevant post-creation.

9.2.2 Reliance & non-Transitive Relationship

The *Reliance & non-Transitive* relationship defines a dependency between two resources that is essential for the survival and functioning of one or both resources. Unlike a Requirement relationship, which is only critical during a specific operation, a Reliance & non-Transitive relationship is vital throughout the entire lifecycle of the resources involved.

If the communication between these resources is severed at any point, it can lead to a failed resource state. The relationship is non-Transitive from one resource to another, i.e., one does not control or manage the other, they are independent. A user can directly perform any operation to either of these resources. The resources are not intertwined, but there still exist a Reliance relationship between them, crucial for their survival.

For instance, in a Kubernetes environment, a user can create a pod with a secret

attached to its configuration. The secret provides essential data, such as authentication tokens or configuration information, to the pod. If the secret becomes unavailable, the state of the pod will eventually fail.

While the pod and the secret are managed independently within Kubernetes, their reliance relationship is required to keep the state of the pod active.

While geo-distributing a resource involved in a reliance relationship, it is essential for the remote resource to have a direct access to the dependent resource. In Cheops, we manage different instances of an application such as individual Kubernetes spread across sites.

With the collaborations, such as replication, Cheops manages to replicate a resource from one site to another. If we try to geo-distribute a pod involved in a reliance relation, such as a pod attached to a secret, from one site to another, the remote pod will fail if it cannot access the exact same secret attached locally.

Hence, Cheops need to ensure that such a scenario does not occur and handle this type of relationship, by ensuring that the dependent resource is accessible to the remote instance of a resource, by any of the collaboration capabilities it offers.

9.2.3 Reliance & Transitive Relationship

Reliance & Transitive defines an intrinsic dependency, where the lifecycle of different resources is tightly intertwined. Reliance relation, as mentioned before, indicates that for the survival of one resource, another needs to be active and accessible. The same is applicable in this class, the difference is in the transitive relationship.

Transitive implies that any operation, such as CREATE, UPDATE, or DELETE to one resource (B) is directly controlled and managed by another resource B. This relation from A to B, is more tightly coupled, compared to reliance & non-Transitive Relationship. In some cases, B can be called as a sub-resource of A, as the latter controls the other.

If the state of resource A goes to a failure state, B will also result in a failed state, even vice-versa is applicable in many cases. Hence, the class also follows a reliance relation.

This relation has some similarities with the Master-Worker [111] approach in distributed system. In this type of relationship, resource A manages and performs CRUD operations on sub-resources, particularly the CREATE operation. This means that the creation of a sub-resource is typically handled through resource A, ensuring that the lifecycle of B is governed by A. This structure enforces a clear hierarchy and control over sub-resource management.

Often, users are unable to perform CRUD operations directly on sub-resources, as resource A serves as the primary API endpoint for any operation on resource B. This means that actions such as creation, reading, updating, or deletion of B must be handled via A.

For example, in Kubernetes, a user can create a Deployment that manages a set of replicated Pods. When a Deployment is created, it automatically generates the desired number of Pods and ensures they remain active. If the Deployment is updated, the corresponding Pods are updated as well to match the new configuration. Similarly, when the Deployment is deleted, all the Pods it manages are also removed, maintaining the lifecycle and state of the application components in a controlled manner.

If we try to update a Pod externally without updating the Deployment, Kubernetes enforces a rollback to the original configuration defined by the Deployment. While a user can perform operations like updating, creating, or deleting a Pod, Kubernetes maintains consistency by reverting changes based on the Deployment specification. This transitive lifecycle management reflects the inherent dependency in the reliance & transitive relationship, where the state of the sub-resource (Pod) is governed by the parent resource (Deployment).

Similarly, in an OpenStack environment, a Heat Stack represents a resource that encompasses a collection of sub-resources, such as virtual machines (VMs), networks, and storage volumes. The Heat Stack manages these resources as a single unit, illustrating the Reliance & Transitive relationship. When a Heat Stack is created, the associated VMs, networks, and volumes are also instantiated, with their configurations defined by the stack. Likewise, when it is deleted, all dependent sub-resources are destroyed, reinforcing the interconnectedness of the resources.

During geo-distribution, Cheops need to ensure that a child resource in this hierarchical relation, has access to its parent and vice versa. It should either replicate both of them to remote site, or try to connect them explicit, such that relationship is satisfied. If this relationship is not satisfied, it can result in failure of either or both parent and child resource.

9.2.4 Local Relationship

A local relationship occur when a resource is dependent on a specific location or instance of an application. In this type of relationship, a resource remains active only if it relies on the availability of another resource or configuration within the same local environment.

If this location-specific dependency is not met or respected, the resource may fail.

Local dependencies are critical because they ensure that resources remain co-located within the same application instance, thereby keeping the resource active. Unlike other relationships that aim to facilitate geo-distribution by solving dependencies across locations, this relationship restricts or excludes dependency values from being distributed to another site. It reflects the inherent property that certain resources or components must remain within a single site, implying that some resource configurations cannot be geo-distributed.

For instance, in Kubernetes, if a pod is attached a local persistent volumes (PVs), they need to be in the same instance of Kubernetes, such that a local PV must be scheduled on the same node where the volume resides. A local PV, requires a storage location tied to the local node where it is scheduled to store the data. If a user create a pod that relies on a local Persistent Volume (PV) for storage, it creates a dependency between the pod and the specific node hosting the local PV. For the pod to remain active, the attached PV must be accessible, and this local dependency prevents the pod from being geo-distributed, as the storage is bound to a specific location.

Similarly, in OpenStack, for a VM to attach to an IP address, both must exist within the same instance, as the IP is allocated from the network, and each network has its own IP range, which can be difficult to replicate across instances. When a user creates a VM in an OpenStack environment, the Neutron service assigns it an IP address from the network resource, which manages IP allocation dynamically. This IP allows the VM to communicate within the network, with the address automatically assigned from a specific subnet by the Neutron service.

If a VM is replicated across different OpenStack instances, the IP address assigned in one instance may not be the same in the remote instance, since each network assigns unique IP addresses. As a result, the configurations of the VM replicas can diverge, especially with respect to their networking setups. When Cheops replicates an operation to the VM, it can send the value of one IP address to another, resulting in conflicting values at different instances.

Hence, local dependency can affect the geo-distribution of a resource to a new site (for instance, if the state of a resource is replicated) or synchronization (specific for replication collaboration, as a value will be different at each site, yet they will be converged).

By omitting this configuration value during these processes, Cheops can ensure that even when different IP addresses are assigned across multiple instances, the resources are

still be considered converged. This approach allows the IP address to remain unique to each instance, preventing Cheops from copying it during an operation and also, maintaining the independent configuration of a VM at each site, in a geo-distributed environment.

9.2.5 Cascading relations

A relationship can belong to any of the previously mentioned types, but a combination of relationships occurs for a resource. For instance, in Kubernetes, a deployment creates a pod (Reliance & Transitive relation) that also has a dependency on a secret (Reliance & Non-Transitive relation). This creates a Cascading dependency structure.

This can result in cascading dependencies where resource A depends on B, which depends on C. During geo-distribution, we must account for dependencies from related resources such as B and C, as a missing link in this chain can cause A to fail. But how can Cheops understand mapping of relations to resources and these cascading ones? We will discuss it along with our solution to identify a relationship.

9.3 How can Cheops identify these relationships?

During geo-distribution, the dependencies, which include requirement, reliance & transitive, reliance & non-transitive and local, must be handled to ensure that the resource will function across multiple instances. The intricacy of these relationships can lead to potential failures of resources if not managed.

We need an approach to map resource dependencies to Cheops in a way that allows it to analyze and verify that all dependencies are met during geo-distribution. Cheops could then ensure that each resource and its dependencies are correctly distributed and active, preventing failures caused by incomplete or broken dependencies, which is important to ensure the illusion of a single application.

To address these challenges, we propose a matrix-based solution, which provides a structured and accessible approach for Cheops to manage resource dependencies in a geo-distributed environment called *relationship matrix*.

The matrix-based solution provides a systematic way to represent interdependencies between resources in an application. By aligning resources along both rows and columns, users can easily input relationships into Cheops, with each cell representing the type of relationship between corresponding resources. This approach simplifies dependency

mapping, making it easier for Cheops to analyze and ensure that all dependencies are accounted for during geo-distribution.

A *relationship matrix* is instantiated for an application and the user provides values before geo-distributing a resource. The matrix is dynamic, allowing values to be updated at any time and users can define any dependencies between resources for any application. This approach offers flexibility, enabling users to adapt and enforce dependencies as needed, ensuring that Cheops handles resource relationships effectively for geo-distribution. A copy of this matrix is sent to the Cheops instances at required sites, before geo-distributing a resource.

Cheops adheres to the principles of being generic and non-intrusive to any application. In cases like the Kubernetes pod example, where secrets are embedded within the pod configuration, identifying dependencies can be challenging while maintaining these principles. The matrix approach, though requiring additional input from the user, aligns with Cheops goals. It provides Cheops with the necessary information without modifying the application business logic, ensuring dependency management without being intrusive. Let us look at this approach in detail.

9.3.1 Matrix based solution for handling dependencies

The matrix-based approach offers a structured method for mapping relationships between resources, systematically showing how each resource interacts with others. This comprehensive overview is critical for Cheops to identify dependencies within an application. Once these relationships are identified, Cheops can efficiently resolve them by taking the necessary actions, ensuring all dependencies are properly addressed during geo-distribution.

Once a matrix is created for an application, it can be reused or replicated to any instance of the application. Let's look at an example of how we construct a matrix with a Kubernetes application:

Matrix Initialization and relationship identification: an example on K8s Application

We consider common K8s resources to create our matrix: Pods, Persistent Volumes (PVs), Persistent Volume Claim (PVCs), ConfigMaps, Secrets, Services, Ingresses, Deployments and ReplicaSets (there are more resources, but we select these in particular for our demonstration.). A relation between these resources can significantly impact their functioning,

as discussed before and must be addressed before geo-distribution.

A square matrix is constructed where both the first row and column lists all the resources in an application. This setup creates a grid frame where each cell represents the type of relationship between the intersecting resources. The diagonal cells, where a resource would relate to itself, are left blank or marked as non-applicable (self dependency for a resource is not considered here), simplifying the matrix as illustrated in Table 9.1.

The matrix is created when the user sends a Cheops CLI command to one of the agents, which then replicates the matrix across all the involved instances of Cheops. This matrix is stored in the Cheops database. For this set of K8s resources, a user can send a request: `Kubectl create pod R --scope {`*Site* 1 `%` *Site* 2`} --relationship_matrix rm.json --relationship_logic rl.go`.

The matrix is sent once to the Cheops agent and it is reused for all of the resource within an application. The *relationship logic* is another concept that I will explain in the later part of this section. It is also sent along with the matrix and scope.

Defining and populating relationships in the matrix

A matrix is constructed by the DevOps, and here, I will explain what the construction process of a matrix looks like for Openstack and K8s. The relationship between resources are defined and categorized into four main types:

1. **Requirement (R)**: Indicates that the resource in the row follows a *Requires* relation with the one in the column.
2. **Reliance & Transitive (RT)**: Indicates that the resource in the row follows a *Reliance & Transitive* relation with the one in the column.
3. **Reliance & non-Transitive (RnT)**: Indicates that the resource in the row follows a *Reliance & non-Transitive* relation with the one in the column.
4. **Local (L)**: Indicates that the resource in the row follows a *Local* relation with the one in the column

To populate the matrix, each cell is filled with the appropriate relationship, based on the dependency between the resource in the corresponding row and column. For example, if a pod has a relation with a local persistent volume (PV) for storage, the cell at the intersection of the pod and the PV is marked with an "L", to indicate a local dependency

	Pod	**PV**	**PVC**	**ConfigMap**	**Secret**	**Service**	**Ingress**	**Deployment**	**ReplicaSet**
Pod	-	L	L & RnT	RnT	RnT	R	R	RT	RT
PV	L	-	RT	-	-	-	-	-	-
PVC	L & RnT	RT	-	-	-	-	-	-	-
ConfigMap	RnT	-	-	-	-	-	-	R	-
Secret	RnT	-	-	-	-	-	-	R	-
Service	R	-	-	-	-	-	RT	R	-
Ingress	R	-	-	-	-	RT	-	-	-
Deployment	RT	-	-	R	R	R	-	-	RT
ReplicaSet	RT	-	-	-	-	-	-	RT	-

Table 9.1: Relationship matrix for Kubernetes application

	VM	**IP**	**Network**	**Volume**	**SecGroup**	**FloatIP**	**LoadBalancer**	**Image**	**HeatStack**
VM	-	L	RT	R	R	L	R	R	RT
IP	L	-	-	-	-	L	-	-	-
Network	RT	-	-	-	-	-	RT	-	RT
Volume	R	-	-	-	-	-	-	-	-
SecGroup	R	-	-	-	-	-	-	-	-
FloatIP	L	L	-	-	-	-	-	-	-
LoadBalancer	R	-	RT	-	-	-	-	-	RT
Image	R	-	-	-	-	-	-	-	R
HeatStack	RT	-	RT	-	-	-	RT	R	-

Table 9.2: Relationship matrix for Open stack application

as illustrated in Table 9.1. This provides a structured approach to map relations between any resource.

Another example with an OpenStack application, for resources such as Virtual Machines (VMs), IP Addresses, Networks, Storage Volumes, Security Groups, Floating IPs, Load Balancers, Images, and Heat Stacks is illustrated in Table 9.2.

Relationships matrix and configuration

A relationship matrix portrays the dependencies between resources in an application, but how can Cheops map or identify these dependencies in an application? For example, in Kubernetes, a pod has a relationship with a secret, which can be identified by the relationship matrix. However, the challenge for Cheops is determining the specific secret

involved (i.e., the identity of the secret). Cheops would need additional information or input from the user to correctly map the secret's identity, ensuring it can recognize and manage this dependency during geo-distribution.

An approach to identifying dependent resources would be to analyze the configuration of the resource itself. A resource is essentially an entity or object that represents a specific component of an application. These resources are defined, managed, and configured through structured configurations, typically written in formats like YAML or JSON. The configuration not only defines the resource itself but also its relationships that leads to dependencies with other resources. In our approach, understanding the structure of these configurations is critical for ensuring that dependencies are resolved during geo-distribution.

By examining the resources configuration details, Cheops can automatically detect associated dependencies, such as secrets or other resources. For example, as illustrated in Figure 9.2 (b), the dependency for a pod, from a deployment $my - app$, is present in the configuration, along with a cascading relation for a secret.

The structure of a Kubernetes resource configuration is hierarchical, as illustrated in Figure 9.2 (b). For a deployment resource, the YAML file specifies the API version and the kind (e.g., Pod, Deployment). This is followed by metadata, such as the resource name and labels.

The spec section then details the specific configuration, including containers, volumes, and environment variables. Within these sections, references to other resources, such as ConfigMaps or Secrets, create relationships that Kubernetes needs to understand and manage during deployment and operation. For instance, a pod may have a volumeMounts section that points to a secret, thereby establishing a relationship with the storage resource.

Although OpenStack, uses a different approach but shares similar concepts in defining and managing resources, as illustrated in Figure 9.2 (a). In OpenStack, resources like virtual machines (VMs), networks, and volumes are often configured using JSON templates, especially when using heat orchestration templates (HOT) to automate infrastructure deployment. The structure of these configurations includes parameters that define resource properties, such as the image for a VM, the network and any attached volumes.

Similar to Kubernetes, OpenStack resources have a hierarchical structure. The top level of an OpenStack resource configuration might define the type of resource (e.g., OS::Nova::Server for a VM), followed by properties such as flavor, image, and key pair.

```yaml
heat_template_version: 2018-03-02

description: >
  Heat stack template to create a simple VM in OpenStack.

parameters:
  vm_name:
    type: string
    description: Name of the virtual machine
    default: my-test-vm
  image_id:
    type: string
    description: ID or name of the image to use for the VM
    default: cirros-0.4.0-x86_64-disk
  flavor_id:
    type: string
    description: Flavor ID or name for the VM
    default: m1.small
  key_name:
    type: string
    description: Name of the SSH key to inject into the VM
    default: my-key
  network_id:
    type: string
    description: ID or name of the network to attach the VM tc
    default: private-network
  security_group:
    type: string
    description: Security group to assign to the VM
    default: default

resources:
  my_vm:
    type: OS::Nova::Server
    properties:
      name: { get_param: vm_name }
      image: { get_param: image_id }
      flavor: { get_param: flavor_id }
      key_name: { get_param: key_name }
      networks:
        - network: { get_param: network_id }
      security_groups:
        - { get_param: security_group }
      user_data_format: RAW
      user_data: |
        #!/bin/bash
        echo "Hello, World!" > /home/ubuntu/hello.txt

outputs:
  vm_name_output:
    description: Name of the created VM
    value: { get_attr: [my_vm, name] }
  vm_private_ip:
    description: Private IP address of the created VM
    value: { get_attr: [my_vm, first_address] }
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: default
  labels:
    app: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:latest
        ports:
        - containerPort: 80
        env:
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: password
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config
      volumes:
      - name: config-volume
        configMap:
          name: my-config
```

Figure 9.2: Sample configuration file for: (a) Heat orchestration template for OpenStack (b) Deployment file for Kubernetes

Dependencies between resources, such as a VM relying on a specific network or security group, are also defined within this structure. These configurations allow OpenStack application to understand how resources should interact and what dependencies must be resolved during deployment.

Cheops needs to analyze the resource configuration to manage dependencies during geo-distribution. The matrix and the configuration need to work in tandem to achieve this.

This dual approach with matrix and configuration, allows Cheops to ensure that an operation from any instance will have the same result across all of the geo-distributed locations, i.e., an illusion of a single application.

A configuration have different ways of expression like Yaml, JSON, CLI, etc., which is based on an application. To understand a configuration, it is important to know about the application context and how these structures function.

For example, a heat orchestration template, as illustrated in Figure 9.2 (a) does not follow the same structure as a deployment configuration illustrated in Figure 9.2 (b). If Cheops tries to understand and learn each of these application configuration structures, it can result in being too specific to an application, how can Cheops be generic?

9.3.2 Relationship logic

The solution we propose is to introduce a *relationship logic* similar to *consistency logic* from Chapter 8 and *resource logic* from Chapter 7. *Relationship logic* is tailored for each application, such as it tries to understand the YAML configuration from Kubernetes or JSON for OpenStack. *Relationship logic* refers to the set of rules and processes that allows Cheops to understand and interpret the resource identifiers without understanding the application itself.

For example, in a Kubernetes deployment, as illustrated in Figure 9.2 (a), Cheops checks the *relationship matrix*, to see what are the possible dependencies that can exist. Cheops then checks whether these dependencies exist for the deployment $my - app$ with the help of *relationship logic*.

The *relationship logic* needs to have a function defined to identify (if it exist) a dependency and the name of it. This will create a list of dependencies with their names, that can be used by Cheops to check if they exist in the remote site during geo-distribution in case of a *Reliance relationship*.

Code 9.1: Relationship Logic for Kubernetes Deployment

```
# Resource type (e.g., Pod, Secret, etc.) refers to the type of
  resources identified from the relationship matrix for a given resource.
 check_dependency_exists(yaml_config, resource_type):
    """
    This function checks if a specific dependency exists in the deployment YAML.
    """
    # Load YAML configuration
    config = load_yaml(yaml_config)

    # Check if the resource type exists in the configuration
    if resource_type in config['spec']:
        return True
    else:
        return False

 action_dependency(yaml_config, resource_type):
    """
    This function extracts the exact name of the dependency from
    the deployment YAML.
    It identifies the specific field where the resource is defined.
    """
    config = load_yaml(yaml_config)

    # Check if the resource type is present and extract its name
    if resource_type == "Pod":
        return config['spec']['template']['metadata']['name']
    elif resource_type == "Secret":
        return config['spec']['template']['spec']['volumes'][0]['secret']
                     ['secretName']
    elif resource_type == "PVC":
        return config['spec']['template']['spec']['volumes'][0]
        ['persistentVolumeClaim']['claimName']
    else:
        return None
```

This *relationship logic* in general would include code for Cheops to understand where to look for dependencies within a configuration file.

- In Kubernetes example from Figure 9.2 (a): the *relationship logic* would involve understanding YAML structure and identifying sections like *envFrom*, *volumes* and *spec* to locate, and analyze dependencies from a deployment.

- In OpenStack example from Figure 9.2 (b): the *relationship logic* would parse JSON configurations to identify relationships between VMs, network, and storage volumes, and understanding the identifiers from the JSON keys and values, defined in the heat stack.

A pseudocode example of *relationship logic* to geo-distribute a deployment resource in Kubernetes application is illustrated in Code 9.1.

The *check_dependency_exists* function is designed to check whether a specific dependency exists in the YAML configuration of a Kubernetes deployment. The function takes two inputs: the YAML configuration file (yaml_config) and the type of resource (resource_type). Resource type (e.g., pod, secret, etc.) refers to the type of resources identified by Cheops for deployment, from the relationship matrix.

Cheops loads the YAML file, then checks whether each of the specified resource type is present in the *spec* section of the deployment configuration. If the resource type is found, the function returns True, indicating that the dependency exists. Otherwise, it returns False.

This function is essential for ensuring that the required dependencies between resources, as defined in the relationship matrix, are present in the actual configuration.

Another function called *action_dependency*, identifies the exact name of the dependency in the local Kubernetes application from the deployment YAML configuration. It works by first loading the configuration and then searching for the relevant resource type within the YAML.

For different resource types, the function looks in different sections of the configuration. For instance, it retrieves the *pod* name from the metadata section, the *secret* name from the volumes section, and the PVC name from the same *volume* section but under the $persistentVolumeClaim$ field.

After identifying a *Reliance & Transitive dependency*, such as a *pod* in a deployment, Cheops ensures that the same hierarchical structure is replicated at the remote site after the deployment is geo-distributed. In contrast, for a *Reliance & Non-Transitive dependency*, like a *secret* in the deployment (which is also a cascading dependency from a pod), Cheops performs a pre-check to verify if the dependent resource is available at the remote site. If it is not available, Cheops will return an error to the user and halt the operation with a message detailing that the relation is not satisfied at the remote site.

Solving dependencies for Local relations

As discussed earlier in Section 9.2.4, handling a *Local relation* is critical to ensure the correct function while geo-distributing a resource across an application. Unlike *Reliance & Transitive relation*, where Cheops needs to check if a dependent resource exist at remote site, here, in this relation, it needs to ensure that the local configuration values are not replicated with any operation.

Our proposed approach to solve such a relation, includes first identifying if such a

Code 9.2: Relationship Logic for Excluding Local Paths in Kubernetes Deployment

```
# Local Path Exclusion Logic for Cheops

check_dependency_exists(path):
"""
This function checks if the given volume mount path is local to the cluster
and should be excluded from the geo-distribution process.
"""
# List of paths considered local to the site and should be excluded
local_paths = ["/etc/config", "/var/data", "/local/path"]
if path in local_paths:
  return True
else:
  return False

action_dependency(resource_config):
"""
This function excludes local paths (like /etc/config) from the deployment
configuration before geo-distribution. It removes the local paths from the
volumeMounts section in the YAML.
"""
# Check if the volumeMount path is local and exclude it
for volume in resource_config['spec']['template']['spec']['containers'][0]
                              ['volumeMounts']:
  if check_dependency_exists(volume['mountPath']):
      print(f"Excluding local path: {volume['mountPath']}")
      resource_config['spec']['template']['spec']['containers'][0]
                     ['volumeMounts'].remove(volume)

return resource_config
```

relationship exist between resources from the *relationship matrix*, as illustrated for K8s Table 9.1 and Openstack Table 9.2 application. Once Cheops identifies the *Local relations* attached to a resource, it processes each dependency, defined by the associated *relationship logic*. The *relationship logic* excludes certain values in a resource configuration before they are geo-distributed. This approach is tailored to specific resource types, and I will illustrate how to build the necessary *relationship logic* using examples from both Kubernetes and OpenStack applications.

For example, let us take the same example as before, for a deployment resource in Kubernetes. As illustrated in Figure 9.2 (b), the deployment consist of a volumeMount attached to a node in the local cluster containing the path $/etc/config$. This path and the value inside it is local to the site and may differ between clusters; replicating this path to another instance of Kubernetes, can cause inconsistencies. This breaks the illusion of having a single application across multiple Kubernetes cluster, as it could fail the geo-distribution of the deployment resource.

To address this challenge, we add a logic to exclude the values that are local for an application instance in the *relationship logic*, as illustrated in Code 9.2.

The function *check_dependency_exists* is designed to determine whether a specific path in the Kubernetes deployment is local to the site and therefore should be excluded during replication. This function uses a predefined list of local paths, such as $/etc/config$, $/var/data$, and $/local/path$, which are known to be site-specific. Each request to geo-distribute a resource is analyzed at the Cheops level, before sending it to any application instance.

The function checks if the given path exists in the configuration for any request and returns True if the path is local, meaning it should be excluded from geo-distribution. If the path is not in the list, the function returns False, allowing the request to continue. This function helps Cheops identify values that are not supposed to be propagated to other clusters in a geo-distributed application.

The *action_dependency* function is responsible for modifying the Kubernetes deployment configuration by removing local path value from the volumeMounts section before the deployment is replicated to a remote site. It iterates over the volumeMounts in the deployment configuration and uses the *check_dependency_exists* function to check if any of the mount paths are local. If a local path is found, it is excluded by removing it from the configuration. This ensures that only non-local resources are replicated to other clusters, avoiding conflicts that could arise from replicating site-specific paths.

For example, if a deployment mounts a volume at $/etc/config$, this path value is removed before geo-distributing the resource, as it is local to the site. By doing this, Cheops ensures that the replicated deployment can function without issues in other clusters, where the same local configuration may not exist or could be different. The user can give a different value for the volume mount at the new site.

Another example of *relationship logic* for an openstack application, where a user tries to replicate a Heat stack configuration with a fixed IP address, from one site to a remote one, is demonstrated in Code 9.3.

The *check_dependency_exists* function is designed to check whether a specific IP address has been assigned to a Virtual Machine (VM) in the Heat Stack configuration. This function identifies local resources, such as IP addresses, that should not be replicated to a remote site. The function works by inspecting the networks section of the VM configuration, specifically looking for the fixed_ips field, which contains the IP address associated with the VM.

If this field exists, the function returns True, indicating that the IP address is present and needs to be excluded from replication. If the fixed_ips field is missing, the function

Code 9.3: Relationship Logic for Excluding IP Address from Openstack Heat Stack

```
# IP Address Exclusion Logic for Heat Stack VM Configuration

check_dependency_exists(vm_config):
    """
    This function checks if an IP address is present in the VM configuration
    from the Heat Stack.
    """
    # Check if 'fixed_ips' field exists and contains the IP address
    if 'fixed_ips' in vm_config['properties']['networks'][0]:
        return True
    else:
        return False

action_dependency(vm_config):
    """
    This function excludes the IP address from the VM configuration by removing
    the 'fixed_ips' field from the networks section.
    """
    if check_dependency_exists(vm_config):
        # Remove the 'fixed_ips' field to exclude the IP address
        del vm_config['properties']['networks'][0]['fixed_ips']
        print("IP address excluded from the configuration.")

    return vm_config
```

returns False, meaning that no local IP is assigned, and no further action is required. This preliminary check ensures that Cheops can correctly identify local resources, preventing potential conflicts when replicating VMs to remote OpenStack instances.

The *action_dependency* function is responsible for modifying the VM configuration by removing the IP address, making it suitable for replication to a remote site. It builds on the result from *check_dependency_exists*. If the IP address is found (i.e., the check function returns True), the function proceeds to delete the fixed_ips field from the networks section of the VM configuration. This ensures that the IP address, which is specific to the local site, is excluded before replication or any geo-distribution.

By removing the IP address, Cheops ensures that the replicated VM will not carry the local IP, allowing the OpenStack Neutron service at the remote site to dynamically assign a new IP address. Let's look at a more generalized workflow for this approach.

9.3.3 A workflow to ensure an illusion of a single application

In this section, I present a workflow that outlines the proposed approach in Cheops to identify and verify resource dependencies in a geo-distributed Kubernetes application. A user send a request to geo-distribute a deployment, as illustrated in Figure 9.2, to a remote site. This deployment has a dependency *Reliance & Transitive* relationship with

a pod and this pod has *Reliance & non-Transitive* relationship with a secret.

1. **Matrix Identification**:
   - The user need to send the *relationship matrix* and *relationship logic* to Cheops along with the request, if it does not exist with the local Cheops agent.
   - Cheops refers to this *relationship matrix*, which classifies different types of resource dependencies, such as Pods, Secrets, PVCs, and ConfigMaps.
   - The matrix indicates potential relationships between the deployment and other resources (pod and replicaset). It indicates the type of relationship (in this case *Reliance & Transitive*, for both dependencies) for each resource.
2. **Configuration Analysis**:
   - After identifying potential dependencies from the *relationship matrix*, Cheops parses the Kubernetes deployment YAML configuration file to verify the existence of these dependencies, using the *relationship logic*.
   - The YAML file contains detailed configurations for resources such as pod and replicaset. Cheops examines specific sections (e.g., 'spec', 'envFrom', 'volumes') to locate resource dependencies defined in the configuration, as programed in *relationship logic*.
3. **Dependency Mapping**:
   - Once dependencies are found from the configuration, Cheops extracts the exact names of the dependent resources, again using the methods defined in the *relationship logic*.
   - Cheops identifies the nature of relations, such as, *Requirement, Reliance & Transitive, Reliance & non-Transitive or Local* and chooses the required *relationship logic* for the dependencies.
   - Cheops examines specific YAML fields, such as 'spec' for replicaset and pod name, 'envFrom' for Secrets and 'volumes' for PVCs and secrets, to extract the relevant names and map them to their dependencies.
4. **Cascading Dependencies**:

- Cheops recursively analyzes each dependent resource found (e.g., Pod, Secrets, PVCs) for additional dependencies.
- For every new resource identified, Cheops repeats steps 1-3 to identify if the new resource has further dependencies. Here, the deployment depends on a pod, Cheops checks if the pod has its own dependencies, such as ConfigMaps or Secrets (which exist here), and maps those as well.

5. **Dependency Verification**:

   - Once Cheops has mapped all dependencies and cascading dependencies, it checks whether each dependent resource is available at the intended remote site for geo-distribution.
   - If any dependencies are missing, Cheops will alert the user to resolve the missing resources. Here, if the *Reliance & Transitive* relations (secret) are missing, it will alert the user and return an error back to user, informing that the operation cannot be completed due to the missing dependency.

9.3.4 Limitation

Although the *relationship logic* introduced in this chapter offers an effective solution for managing dependencies in geo-distributed environments, it requires developers or DevOps to have a thorough understanding of the application configuration files. While we successfully identified and resolved dependencies while maintaining the Cheops non-intrusive approach, i.e., without altering the application business logic, users are still required to manually interpret and program the entire configuration into the *relationship logic*. This process can be labor-intensive and prone to errors, particularly for complex applications with numerous dependencies and intricate configurations.

9.4 Validation

We validate the relationship model to ensure the illusion of a single application with Kubernetes across multiple geo-distributed sites. We used four combination of geo-distributed resources within Kubernetes, to demonstrate the four types of relations: Requires, Reliance & Transitive, Reliance & Non-Transitive, and Local.

The validation involved geo-distributing a set of Kubernetes resources (Pods, Secrets, Persistent Volumes, ConfigMaps, etc.) across two geographically dispersed Kubernetes clusters at *Site* 1 ans *Site* 2, managed by Cheops. The experimental setup is the same as described in Section 6.3. The goal is to ensure that Cheops can accurately handle resource dependencies, allowing the illusion.

The relationship matrix for Kubernetes application was already defined in Table 9.1, we use this matrix to identify the dependencies along with the relationship logic, as defined in Listing 9.4. Let us look at each of the relationship type within the Kubernetes application:

Managing Requires Relationship

A *Requires* relationship exists between a Pod and a Service in a Kubernetes application, as shown in the K8s *relationship matrix* Table 9.1. This is a temporary dependency where the Pod needs the Service for specific operations, such as network communication or exposure to the outside world. This dependency is only relevant while the Pod is actively interacting with other services or clients.

A Service provides a stable IP address or DNS name, allowing external users or other services within the cluster to communicate with the Pod.

In this scenario, we deploy a Pod and its corresponding Service using Cheops. At *Site* 1, the operations `cheops --cmd kubectl create pod foo -f pod.yaml --scope Site 1` and `cheops --cmd kubectl create service foo-svc -f svc.yaml --scope Site 1` are applied, deploying the Pod foo and the Service $foo - svc$ at $Site1$. The Service is connected to the Pod via its configuration.

Next, we attempt to geo-distribute the Pod foo to *Site* 2 by executing the operation `cheops --cmd kubectl create pod foo -f pod.yaml --scope Site 1 & Site 2`. Cheops detects a *Requires* relationship between the Pod and Service, as identified in the *relationship matrix* Table 9.1.

Cheops uses the *relationship logic*, as illustrated in Code 9.4, to determine the attached Service name. Cheops performs a check to ensure the Service exists at *Site* 2. If not, it notifies the user to maintain the illusion of a unified application across geo-distributed instances. If the Service is not available at *Site* 2, the user can access the Pod foo via DNS at *Site* 1, but not in *Site* 2.

Code 9.4: Relationship Logic for a Pod

```
# Resource type (e.g., Service, Secret, PVC, etc.) refers to the type of
# resources identified from the relationship matrix for a Pod resource.
check_dependency_exists(pod_yaml_config, resource_type):
"""
This function checks if a specific dependency exists in the Pod YAML configuration.
"""
# Load YAML configuration
config = load_yaml(pod_yaml_config)

# Check if the resource type exists in the Pod configuration
if resource_type == "Service":
  # A service is indirectly linked to a Pod through labels and selectors
  return "labels" in config['metadata'] and "app" in config['metadata']['labels']
elif resource_type == "Secret":
  # Check if a Secret is mounted in the Pod
  return any("secret" in vol for vol in config['spec']['volumes'])
elif resource_type == "PVC":
  # Check if a PVC is mounted in the Pod
  return any("persistentVolumeClaim" in vol for vol in config['spec']['volumes'])
elif resource_type == "ConfigMap":
  # Check if a ConfigMap is mounted in the Pod
  return any("configMap" in vol for vol in config['spec']['volumes'])
elif resource_type == "ReplicaSet":
  # ReplicaSet is not stored directly in the Pod YAML
  return False
else:
  return False


action_dependency(pod_yaml_config, resource_type):
"""
This function extracts the exact name of the dependency from the Pod YAML configuration.
It identifies the specific field where the resource is defined.
"""
config = load_yaml(pod_yaml_config)

# Check if the resource type is present and extract its name
if resource_type == "Service":
  # The Service name can be inferred from the Pod's labels
  return config['metadata']['labels']['app']
elif resource_type == "Secret":
  # Extract the name of the Secret used by the Pod
  for vol in config['spec']['volumes']:
      if "secret" in vol:
          return vol['secret']['secretName']
elif resource_type == "PVC":
  # Extract the name of the PVC used by the Pod
  for vol in config['spec']['volumes']:
      if "persistentVolumeClaim" in vol:
          return vol['persistentVolumeClaim']['claimName']
elif resource_type == "ConfigMap":
  # Extract the name of the ConfigMap used by the Pod
  for vol in config['spec']['volumes']:
      if "configMap" in vol:
          return vol['configMap']['name']
else:
  return None
```

Managing Reliance & Transitive Relationship

A $Reliance\ \&\ Transitive$ relationship in Kubernetes exists between a Deployment and its Pods, where the Deployment manages the Pod lifecycle. Any changes to the Deployment directly affect the Pods, ensuring they maintain the desired state.

In this validation, we deploy a Deployment *bar* that manages three Pods with Cheops. At *Site* 1, the operation `cheops --cmd kubectl create deployment bar -f dep.yaml --scope Site 1` is executed, which automatically creates three Pods as per the Deployment configuration.

Next, we geo-distribute the Deployment *bar* to *Site* 2. The operation `cheops --cmd kubectl create deployment bar -f dep.yaml --scope Site 1 & Site 2` replicates the Deployment across both sites. Prior to applying the operation, Cheops identifies the *Reliance* & $Transitive$ relationship from the *relationship matrix*, portrayed in Table 9.1.

Since the Pod configurations are embedded in the Deployment, Cheops does not handle the Pods separately. Instead, it informs the user that transitive resources will also be deployed at *Site* 2. Once the Deployment is created, Kubernetes automatically spawns the necessary Pods. This ensures the illusion of a unified application across geo-distributed instances is maintained.

Managing Reliance & Non-Transitive Relationship

A $Reliance\ \&\ Non - Transitive$ relationship exists between a Pod and a Secret in Kubernetes, where the Pod relies on the Secret for credentials or configurations but does not manage it.

In this validation, we deploy a Pod *bar* that relies on a Secret *biz* for authentication. At *Site* 1, the Secret *biz* is already deployed, and the operation `cheops --cmd kubectl create pod bar -f pod.yaml --scope Site 1` is executed, creating the Pod *bar* that references *biz*.

Next, we geo-distribute this Pod to *Site* 2. The operation `cheops --cmd kubectl create pod bar -f pod.yaml --scope Site 1 & Site 2` replicates the Pod. Prior to executing the operation, Cheops identifies the $Reliance\ \&\ Non - Transitive$ relationship between the Pod and the Secret from the *relationship matrix* portrayed in Table 9.1.

Cheops checks the relationship logic, illustrated in Code 9.4, to extract the Secret name, *biz*. If the Secret is not available at *Site* 2, Cheops returns an error, indicating that replication cannot proceed until the Secret is replicated or manually created. This ensures that the Pod will function correctly and preserves the illusion of a unified application

across geo-distributed instances.

Managing Local Relationship

A *Local* relationship exists between a Pod and a Persistent Volume (PV) in Kubernetes, representing a location-specific dependency.

In this validation, we deploy a Pod *bar* that relies on a local PV *vol* for storage. At *Site* 1, the PV is already provisioned, and the operation `cheops --cmd kubectl create pod bar -f pod.yaml --scope Site 1` is executed, creating the Pod *bar* bound to the local PV.

Next, we attempt to geo-distribute the Pod to *Site* 2. The operation `cheops --cmd kubectl create pod bar -f pod.yaml --scope Site 1 & Site 2` replicates the Pod, but Cheops detects that the PV cannot be replicated to *Site* 2 due to its local nature, identified by the matrix Table 9.1. Cheops identifies the name of the PV with the logic described in Code 9.4 and checks *Site* 2 for the dependent resource. If it is not available, Cheops notifies the user to configure a new storage solution at *Site* 2 or the operation will not proceed.

This ensures that local storage dependencies are not propagated incorrectly across sites, maintaining the illusion of a unified application while preserving site-specific configurations.

9.5 Summary

In this chapter, we explored the complexities of managing dependencies in geo-distributed applications and introduced a structured approach to handling these dependencies externally within the Cheops framework. The inherent interdependencies pose significant challenges when resources are distributed across geographically dispersed sites, if they are not resolve, it can lead to failure of the resources within an application. The relationship model and matrix solution presented here, categorize these dependencies into four key classes: Requires, Reliance & Transitive, Reliance & Non-Transitive, and Local, offering a clear distinction for understanding and managing the intricate links between resources.

The relationship matrix plays a crucial role in this process, providing a systematic and visual representation of the interdependencies within an application. This matrix not only identifies but also maps the relationships between resources, enabling Cheops to ensure that all necessary resources are available and correctly configured before proceeding with any geo-distribution operations. By integrating this matrix into Cheops, we enable the

orchestrator to automate the geo-distribution, ensuring that all dependencies are met and maintained across different sites. This approach significantly reduces the risk of errors and ensures that the applications remain stable and functional after geo-distribution.

Furthermore, the introduction of a relationship logic, tailored to the specific configuration formats of applications like Kubernetes and OpenStack, ensures that Cheops remains non-intrusive and generic. This logic allows Cheops to interpret and analyze resource configurations without needing to understand the application context directly, thereby preserving its core principles. The exclusion list mechanism within this logic further enhances Cheops capability to manage local dependencies effectively, ensuring that site-specific configurations do not hinder the geo-distribution process. We validate the approach with resources in Kubernetes application.

In conclusion, the methodologies and solutions discussed in this chapter significantly enhance Cheops ability to manage geo-distributed microservices. By systematically addressing the complexities of resource dependencies, we ensure that Cheops can ensure that the resources will remain in active state after geo-distributing them.

Conclusion and Perspectives

The growing demand for real-time data processing close to the user, fueled by the rise of Industry 4.0 applications like Virtual Reality, Augmented Reality, and others, highlights the importance of geo-distributed applications. Current methods that enable geo-distribution of applications often require intrusive modifications in their business logic, making them difficult to implement in existing systems.

A key focus of this research has been to create an approach to geo-distribute any application without requiring substantial changes to their original code. This thesis works on the existing Cheops solution, introduced by the STACK research group, offering a non-intrusive and generic solution to geo-distribute an application. This approach decouples geo-distribution concerns from the core application, making it possible to distribute resources and operations across multiple geographic locations.

In my thesis, I identified three limitations with the existing Cheops solution:

1. **Challenges with Replication**: While replication is widely used in systems like CDNs and databases, it introduces challenges such as high synchronization overhead, as every operation must be performed at each site. These issues can result in network congestion, especially at the edge where resources are constrained, leading to delays and inefficient resource utilization in geo-distributed environments.

2. **Limitations of RAFT-based Consensus**: Cheops employed a RAFT-like consensus mechanism, but it encountered difficulties during network partitions, as reaching quorum took longer, potentially causing multiple operations to roll back. The rollback process is complex, requiring Cheops to store the resource's state before each operation. This adds the challenge of Cheops needing to understand the application context and operations to execute the rollback properly.

3. **Difficulty in Maintaining the Illusion of a Unified Application**: Cheops strives to create the illusion of a seamless, unified application running across geo-distributed instances, ensuring that any operation working on a single instance also functions in a Cheops environment. However, issues arise when resource dependencies are not captured across instances. If a dependency is unavailable at a remote

site, the geo-distributed resource may fail. In contrast, in a local instance, dependencies are typically available with the resource, and this inconsistency can undermine the illusion of a unified application across multiple instances.

This led to the creation of three different research questions:

1. Can we develop an alternative collaboration method that addresses the challenges of replication while adhering to Cheops principles of local-first, non-intrusive and application-agnostic design?

2. Is it possible to conceptualize a consistency approach that overcomes the current limitations of Cheops, while staying aligned with its core values of being local-first, non-intrusive, and broadly applicable?

3. How can we ensure that operations within a single instance are successfully replicated across a Cheops geo-distributed environment by resolving dependency issues, thereby preserving the seamless illusion of a unified application?

This thesis addresses these questions by introducing further components into the existing Cheops solution.

Figure 9.3: Cheops components

Figure 9.3 illustrates the core components of Cheops. It already included $Local-first$, which prioritizes operations during network partitioning, and two collaboration models $Sharing$ and $replication$. In my thesis, I expanded the approach with:

- **Cross collaboration model** is a novel approach to resource distribution that is based on sharding rather than traditional replication. This reduces the synchronization overhead typically associated with full replication, especially in environments with limited bandwidth and computing resources, such as the edge.

- **Externalizing Consistency** is a new approach that externalizes consistency management, ensuring it remains local-first to any operation, generic and non-intrusive to any application. This approach can identify and resolve concurrent operations within a geo-distributed application and, in some cases, address conflicts beyond concurrency.

- **Ensuring a unified single application** is achieved by managing dependencies between geo-distributed instances for a resource. We propose an approach to identify and resolve these dependencies in a generic and non-intrusive manner, external to the application logic based on the relationship model we defined.

- **Dedicated API** offers a specialized interface tailored to Cheops geo-distribution requirements and operations while being generic and non-intrusive to any application.

An observation made from this thesis is that an application can be viewed as a collection of different logics, with the **business logic** being the primary or foundational one. In order to achieve a generic and non-intrusive geo-distribution approach, we extended this concept by introducing multiple additional logics **Resource logic**, **Consistency logic**, **Hooks logic**, and **Relationship logic**, that operate independently of the existing application logic, as illustrated in Figure 9.4.

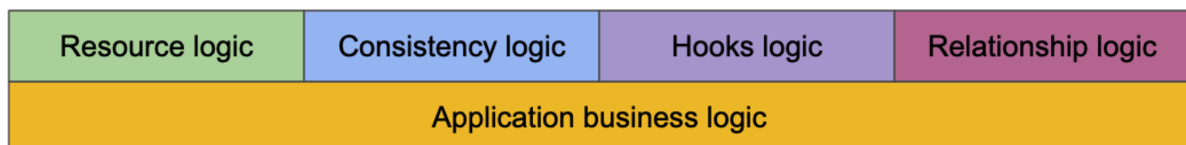

Figure 9.4: Different logics in Cheops

- **Resource Logic**: This layer handles the sharding process in the cross collaboration, ensuring that resources are appropriately partitioned and managed across geo-distributed environments.

- **Consistency Logic**: This logic ensures that the application maintains strong eventual consistency across geo-distributed replicas, resolving concurrent conflicts and ensuring that the distributed states converge.

- **Hooks Logic**: This logic reacts to specific errors or events in an application, ensuring convergence of replicated resources beyond concurrent operations.

- **Relationship Logic**: This logic manages the interdependencies and relationships between different resources during geo-distribution.

These externalized logics handle different aspects of the applications distributed behavior without interfering with or altering the core business logic. By isolating these responsibilities, the approach remains modular and scalable, ensuring that the complexity of geo-distribution is abstracted from the core application, enabling smoother integration and maintenance across distributed application.

Perspectives

The Cheops approach has demonstrated that geo-distribution can be achieved in a generalized and non-intrusive manner, while ensuring local-first operations. However, several areas present opportunities for further research and development to enhance its functionality, scalability, and adaptability. Below are key areas for future work, based on identified needs and potential improvements:

Perspective 1: Automated Error Management: Control Loops Based on Collaboration

A promising direction for future work is the implementation of automated error management in the Cheops framework, leveraging control loops inspired by the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop [112]. Given that all resources in Cheops are deployed through a collaboration mechanism, such as Sharing,

Replication or Cross, these collaborations represent critical points where errors may occur. Integrating control loops into each collaboration type would allow for automated monitoring and handling of failures, improving system reliability and reducing the need for manual intervention.

In the MAPE-K loop, the system continuously monitors the state of resources and collaborations, analyzes any deviations from expected behavior, plans a corrective action based on predefined rules, and then executes the necessary adjustments. This process is supported by a shared knowledge base, which stores historical error data and resolution strategies, allowing the system to learn and optimize its responses over time.

Partial error capture and relay any errors that occur during a request back to Cheops, providing detailed information about the nature of the error at each involved site. Rather than allowing a single failure to disrupt the entire geo-distributed application, Cheops can isolate these partial errors and apply targeted recovery actions, such as retrying the operation or rolling back the affected resource.

The ability to capture detailed information about these errors—such as the type of failure, location, and timing—provides valuable insights that can be fed into the knowledge base of the MAPE-K loop. This knowledge base, enriched with historical error data, enables Cheops to learn from past partial errors and optimize future responses. Over time, it could also enable predictive error detection, allowing Cheops to foresee and prevent similar errors before they occur, enhancing the overall resilience of the geo-distributed application.

The approach will be similar to the existing Kubernetes concept of a *replica set*, where a pod is continuously monitored, and if its state deviates from the desired state, the Kubernetes API server restores it.

Control Loop Mechanism Based on Collaboration

Each collaboration method in Cheops would be equipped with its own control loop, customized to address the specific challenges it faces:

- **Monitor**: The control loop continuously monitors the health and status of resources involved in a collaboration. For instance, in a replication scenario, it would track whether all replicas are successfully synchronized across sites, checking for signs of failure such as network timeouts, communication failures, or resource inconsistencies.

- **Analyze**: Upon detecting an anomaly, such as a failed replication or missing re-

source, the control loop analyzes the cause and classifies the error. Based on the nature of the failure, from the *partial error*, it can distinguish between transient errors (e.g., brief network outages) and persistent errors (e.g., configuration mismatches), as well as critical errors that may require immediate rollback.

- **Plan**: Depending on the error type, the control loop formulates an appropriate response. For example, if the error is transient, the system could plan a simple retry. In the case of persistent errors, it may plan a rollback to a stable state or initiate further diagnostics. For critical errors, the plan might involve notifying devops or switching to a deterministic fallback process.

- **Execute**: Finally, the control loop executes the planned response. This could involve retrying the failed operation, rolling back to a previous stable state, or triggering an alert if Cheops cannot resolve the issue automatically.

Throughout the process, the Knowledge Base is updated with data on errors, responses, and outcomes. Over time, this historical database enables Cheops to refine its error-handling and even predict failures, creating a self-learning system.

By implementing MAPE-K control loops, Cheops can enhance fault tolerance and resilience, making it a more robust, self-managing platform for geo-distributing applications. This approach improves system reliability and paves the way for smarter, adaptive and non-intrusive geo-distribution in the future.

Perspective 2: Improving the Consistency Approach

In this thesis, I proposed an external consistency management approach for geo-distributed applications, aimed at resolving the challenges associated with traditional consistency protocols, as described in Chapter 8. Despite the advantages of being generic and non-intrusive, while ensuring local-first operations for any application, certain limitations still exist, which present areas for further improvement, as detailed in Section 8.5.

To address these limitations, future work could explore the following areas:

- **Enhanced API Access for Better Operation Capture**: A key area for improvement is extending API access or finding alternative methods to capture all relevant operations, even those not currently exposed. This would allow the consistency management system to handle a broader range of operations more effectively and ensure more comprehensive synchronization.

- **Stricter Logic for Enforcing Correctness**: Future work could focus on developing a stricter consistency logic that incorporates application-specific rules, such as invariant checks or transaction-level consistency enforcement. This would ensure that critical applications like financial systems adhere to strict conditions, preventing cases where the final state, though converged, is incorrect. For example, by applying transactional locks or consensus algorithms, the system could guarantee that concurrent transactions do not violate key invariants, such as ensuring a bank account balance does not drop below zero.

- **Adaptive Consistency Mechanisms**: Another direction could involve creating adaptive consistency mechanisms that dynamically adjust based on application needs and conditions. By allowing the system to switch between different consistency models depending on the current operation criticality, performance and correctness can be optimized for various workloads and scenarios in a geo-distributed environment.

This provides a structured and comprehensive Perspective on improving the current consistency approach, focusing on enhancing the API access, introducing stricter logic for application correctness, and exploring adaptive consistency mechanisms to make the approach more versatile and robust for different use cases.

Perspective 3: Automating Relationship Logic for Simplified Dependency Management

In this thesis, I proposed a structured approach to handling resource dependencies in geo-distributed applications using Cheops, as described in Chapter 9. While the solution successfully manages dependencies without altering the business logic of applications, certain limitations still exist, particularly the requirement for developers or DevOps teams to manually interpret and configure the *relationship logic*, as discussed in Section 9.3.4.

To address these limitations, future work could explore the following areas:

- **Automated Configuration Parsing**: An approach that automates the extraction and mapping of dependencies from application configurations would greatly reduce the need for manual input. By incorporating rule-based or MAPE-K models, the system could automatically analyze and map dependencies, reducing human effort and minimizing the potential for errors.

- **Dynamic Relationship Logic**: Future work could focus on developing a self-adaptive mechanism that dynamically adjusts the *relationship logic* based on real-time analysis of the application dependencies. This would ensure that as the application evolves, the *relationship logic* stays updated without requiring manual reconfiguration, making the system more robust and scalable.
- **Context-Aware Dependency Management**: Another area for improvement is enhancing the Cheops ability to identify and resolve dependencies contextually, based on the specific conditions of the application or the geo-distributed environment. By implementing context-aware rules, Cheops could adapt its *relationship logic* based on different operational scenarios, improving its ability to handle complex configurations.

Summary and (re) opening a question

In conclusion, this research extended the existing Cheops solution to geo-distribute an application without requiring significant modifications to their business logic. We further proposed future enhancements to focus on areas such as automated error management through control loops and refining the consistency model.

By working on logic and incorporating control loops to orchestrate the lifecycle of geo-distribution concerns, we invite the community to reconsider the relevance of aspect-oriented programming [113]. While significant research has been done at the function/process granularity level, particularly in Java applications [114], aspect-oriented programming activities have been relatively quiet in recent years.

With the emergence of middleware solutions like Cheops and the concept of handling multiple concerns through specific logics, the landscape seems ripe for a revival of aspect programming. These tools are already addressing complex concerns in modern distributed systems, demonstrating that the right entry point could be the REST API or similar integration points.

Could this be the right moment to revive aspect-oriented programming and explore its potential for addressing the challenges of geo-distributed systems? Should we revisit this paradigm as a way to modularize and handle cross-cutting concerns in todays complex application environments?

Résumé en français

Le paradigme de l'informatique en nuage (cloud computing) a marqué un changement fondamental dans la manière dont les ressources et services informatiques sont fournis [1, 2]. Il permet aux utilisateurs d'accéder à des ressources informatiques, y compris les infrastructures et logiciels, à la demande, sans avoir à gérer directement ces ressources.

Ce paradigme nécessite un emplacement centralisé dédié à l'hébergement de tout le matériel et des ressources informatiques, appelé centres de données (Data Centers, DC).

Avec la croissance du paysage numérique, le besoin de traitement des données plus proche de leur source a conduit à l'essor de l'informatique de périphérie (edge computing). Le passage actuel aux applications de l'Industrie 4.0, qui représente l'intégration de technologies telles que l'Internet des objets (IoT), l'intelligence artificielle (IA), la conduite autonome, la réalité augmentée et virtuelle (AR/VR), nécessite un traitement et une prise de décision en temps réel plus proches de la source des données.

Cela a entraîné une demande croissante de calcul à proximité de la source des données ou de l'utilisateur, accélérant ainsi la transition de l'informatique en nuage centralisée vers l'edge [3, 4].

L'Industrie 4.0 a non seulement accéléré cette transition vers les dispositifs périphériques, mais a également mis en évidence la nécessité d'une collaboration fluide entre ces environnements informatiques distribués. Cela a conduit à une dispersion géographique du calcul à travers le monde, établissant le paradigme géo-distribué.

Cette transition vers la géo-distribution, où les applications peuvent être réparties sur différentes régions géographiques, est devenue essentielle pour assurer la faible latence, la disponibilité et la résilience [5, 6, 7]. De nombreuses recherches sont menées sur ce sujet, en particulier par le groupe de recherche STACK, au sein duquel j'ai développé les activités présentées dans ce manuscrit.

Les solutions existantes pour la géo-distribution intègrent la logique de distribution directement dans le code de l'application, ce qui signifie que l'application elle-même gère la manière dont les ressources sont réparties et synchronisées entre différents emplacements. Cette approche rend le système intrusif, car elle nécessite des modifications du code de base de l'application.

Par exemple, la fédération d'un service comme Keystone sur plusieurs instances d'application OpenStack, connue sous le nom de *Federated Keystone* [8], nécessite la modification du service existant pour permettre la géo-distribution. De même, les approches middleware comme Kube-Edge [9] requièrent la création de courtiers spécifiques capables d'interpréter l'application afin de géo-distribuer une application Kubernetes entre instances. En outre, les approches de base de données comme AntidoteDB [10] exigent des modifications substantielles du code de l'application pour prendre en charge la géo-distribution.

En conséquence, il devient difficile d'adapter ou d'étendre un système à de nouveaux environnements ou exigences, car chaque changement peut nécessiter la modification du code interne de l'application [11].

Un autre problème avec ces solutions est leur manque de généralité, car elles sont souvent adaptées à une application spécifique, ce qui les rend difficiles à appliquer à différents systèmes. Ces solutions reposent généralement sur une logique ou des API spécifiques à un environnement particulier, limitant ainsi leur adaptabilité. Par exemple, KubeEdge [9] est une solution qui géo-distribue les ressources Kubernetes, mais elle est conçue spécifiquement pour l'application Kubernetes.

Ce manque de généralité signifie que chaque application a besoin de sa propre solution de géo-distribution, ce qui augmente encore la complexité du développement et les coûts. De plus, ces solutions non génériques demandent un effort considérable pour être modifiées lorsque l'architecture de l'application change ou lorsqu'elles sont étendues à de nouvelles régions, car elles ne peuvent pas être facilement réutilisées sur différentes plateformes ou services.

Cette analyse a conduit à la question de recherche actuellement explorée par l'équipe STACK :

Est-il possible de créer une solution pour géo-distribuer n'importe quelle application sans être intrusif au sein de sa base de code existante ?

Ma collègue, Marie Delavergne, a abordé cette même question dans sa thèse de doctorat [12] et a introduit la première version de notre solution, appelée **Cheops**.

Notre première approche pour géo-distribuer une application de manière externe

Comme décrit précédemment, les solutions existantes pour géo-distribuer une application sont soit trop spécifiques, soit nécessitent des modifications du code métier existant. Cela

crée le besoin d'une solution plus générique et non intrusive qui puisse fonctionner avec n'importe quelle application sans altérer sa logique métier.

Pour répondre à ces besoins, le framework Cheops a été développé comme une solution middleware par le groupe de recherche STACK [11, 12, 13, 14]. La solution que je présente ci-dessous s'appuie sur ces travaux existants, en introduisant le concept fondamental de Cheops.

Ce framework permet à n'importe quelle application d'être géo-distribuée sans nécessiter de modifications significatives de sa logique métier. En utilisant un protocole largement adopté comme l'API REST et en se basant sur des API spécifiques à l'application (plutôt que des API personnalisées) pour la communication, Cheops maintient son caractère générique et non intrusif.

Cheops considère un ensemble d'instances indépendantes d'une application (une par site) et orchestre les requêtes selon les besoins des utilisateurs. Il fonctionne selon deux principes principaux : local-first et collaborative-then. Le principe local-first garantit que chaque instance d'application fonctionne de manière autonome. Il permet la géo-distribution des applications en déployant une instance complète sur chaque site, garantissant que chaque site peut gérer les requêtes de manière autonome.

Cependant, lorsque des ressources ne sont pas disponibles localement, Cheops facilite la collaboration entre les instances indépendantes pour partager les ressources entre les sites. Ainsi, le second principe collaborative-then a été introduit, permettant aux instances de collaborer lorsque cela est nécessaire. Ce principe aide les applications à surmonter la limitation des ressources indisponibles localement en transférant et partageant dynamiquement les requêtes entre instances.

Cheops disposait de deux mécanismes de collaboration : le partage (sharing) et la réplication (replication). Le partage permet aux instances d'applications indépendantes de transférer dynamiquement les requêtes à d'autres instances lorsqu'une ressource nécessaire n'est pas disponible localement, permettant ainsi une distribution efficace sans dupliquer inutilement les ressources. Par exemple, si le service de calcul (compute) *nova* dans OpenStack sur un site a besoin d'une image qui est stockée sur un autre site, Cheops peut transférer une requête au site distant pour récupérer la ressource, permettant ainsi de satisfaire la requête locale.

La réplication, quant à elle, crée et gère des copies identiques d'une ressource sur plusieurs instances pour garantir la cohérence et la disponibilité, même en cas de partitions réseau ou de pannes de sites. Cette collaboration garantit que les ressources critiques sont

toujours accessibles sur tous les sites.

Une opération peut être effectuée sur l'une de ces copies identiques depuis n'importe quelle instance d'une application, suivant le principe local-first. Lorsqu'un changement est effectué sur une réplique, il doit être appliqué à toutes les autres répliques pour maintenir la cohérence de la ressource. Cependant, si deux utilisateurs apportent des modifications en même temps à partir de différents emplacements à la même ressource répliquée (appelées opérations concurrentes), cela peut entraîner des incohérences entre les sites. Pour éviter cela, une méthode de synchronisation est nécessaire pour garantir que la ressource reste cohérente sur tous les sites.

Cheops utilisait un protocole de consensus de type RAFT [12] pour garantir une cohérence finale forte. Contrairement au protocole RAFT conventionnel [15], cette approche exécutait initialement une opération localement, dès réception d'une requête de l'utilisateur, avant de chercher à obtenir un consensus par majorité (quorum).

Ensuite, il vérifie si une opération peut obtenir le quorum requis. Si l'opération échoue à obtenir un quorum, un mécanisme de rollback est en place pour annuler les opérations lorsque le consensus n'est pas atteint, garantissant que chaque réplique sera finalement cohérente. Cette approche assure la synchronisation entre les sites géo-distribués tout en préservant la nature local-first de chaque opération. Cependant, elle nécessite que Cheops stocke l'état d'une ressource avant d'appliquer une opération, afin de permettre un mécanisme de rollback si nécessaire.

La première version de Cheops introduit Scope-lang, un langage spécifique au domaine (DSL) qui permet aux utilisateurs de spécifier les exigences de géo-distribution en même temps que leurs requêtes API. Ce langage permet au système de comprendre comment gérer et géo-distribuer les ressources entre plusieurs sites en fonction des instructions des utilisateurs. En conséquence, Cheops peut fonctionner avec l'API native de l'application, tout en ajoutant des instructions Scope-lang à chaque requête, ce qui le rend générique pour toute application.

Grâce à ces principes, Cheops crée l'illusion d'un système unique et unifié en connectant des instances indépendantes d'applications (similaire aux premières recherches sur l'image de système unique (SSI) [16]), permettant une collaboration et un partage de ressources transparents dans des environnements géo-distribués.

Limites de l'approche existante

Bien que l'approche existante offre une solution complète pour géo-distribuer n'importe quelle application tout en séparant la géo-distribution de la logique métier, elle présente encore certaines limites. Voici un aperçu de ces limitations dans l'approche de cette version de Cheops :

1. **Problèmes avec la réplication** : La réplication, bien qu'elle soit couramment utilisée (par exemple, dans les CDN et les bases de données), pose des défis tels que la surcharge de synchronisation due à la nécessité de partager toutes les opérations avec chaque site. Une réplication complète peut ne pas être nécessaire en permanence. Ces problèmes peuvent entraîner une congestion du réseau (surtout à la périphérie, en raison de contraintes), des retards et une utilisation inefficace des ressources dans des environnements géo-distribués.

2. **Limites du consensus basé sur RAFT** : Cheops utilisait un mécanisme de consensus de type RAFT, qui rencontre des difficultés lors d'une partition réseau, nécessitant plus de temps pour atteindre un quorum, ce qui peut entraîner l'annulation de plusieurs opérations. Le processus de rollback lui-même est compliqué, car il nécessite que Cheops stocke l'état d'une ressource avant chaque opération et ajoute des opérations supplémentaires.

3. **Difficulté à créer l'illusion d'une application unique** : Cheops vise à créer l'illusion d'une application unifiée fonctionnant de manière transparente à travers des instances géo-distribuées, ce qui signifie que toute opération possible au sein d'une instance unique doit également être réalisable dans un environnement géo-distribué Cheops. Cependant, des défis surviennent lorsqu'une opération ne parvient pas à capturer une dépendance (où une ressource dépend d'une autre pour fonctionner) entre les ressources. Si cette interdépendance n'est pas satisfaite sur le site distant, la ressource géo-distribuée peut se retrouver dans un état défaillant. En revanche, la probabilité que cela se produise au sein d'une instance locale est plus faible, car les dépendances sont généralement déjà disponibles localement avec la ressource initiale. Cela peut briser l'illusion d'une application unifiée, car des comportements différents sont observés entre une instance unique et une application géo-distribuée Cheops pour la même opération.

Questions de recherche

Les limitations mises en évidence ci-dessus nous ont motivés à identifier trois questions de recherche qui ont orienté ma thèse :

1. Est-il possible de concevoir une méthode de collaboration alternative qui résout les problèmes de réplication tout en respectant les principes de Cheops de priorisation du local-first, de non-intrusivité et d'applicabilité à toute application ?

2. Pouvons-nous conceptualiser une approche de la cohérence qui surmonte les limites de l'approche actuelle de Cheops tout en s'alignant sur ses principes de local-first, de non-intrusivité et d'applicabilité à toute application ?

3. Pouvons-nous garantir que les opérations dans une instance unique peuvent être reproduites avec succès dans un environnement géo-distribué Cheops, créant ainsi l'illusion transparente d'une application unique ?

Contributions

Nous avons publié un **article de recherche** et un court article dans des revues internationales dans le cadre de ma contribution à mon doctorat.

1. L'article de recherche a été présenté à la conférence ICFEC 2024 [17], détaillant un modèle de collaboration novateur appelé Cross.

2. Un court article a été publié lors de la conférence ICSOC 2022 [13], présentant une introduction à la première architecture de Cheops et un aperçu initial de mon travail de recherche.

J'ai fait plusieurs **présentations** sur mes travaux de recherche dans divers forums :

1. J'ai récemment présenté un poster à l'atelier Cloud Control Workshop 2024 [18], où j'ai présenté mon travail et notre vision de Cheops.

2. J'ai fait une présentation orale et un poster à la conférence Compas 2022 [19], où j'ai donné un aperçu initial de mon travail.

3. Nous avons présenté à l'OpenInfra Summit 2022 [20], où nous avons discuté de la première architecture de Cheops à la communauté des développeurs.

4. J'ai donné une présentation orale à la conférence Journées Cloud 2021 [21], où un premier aperçu des travaux existants sur Cheops et des futures directions de recherche a été discuté.

5. J'ai également fait une présentation orale et un poster lors d'une école d'hiver (6e édition de l'école d'hiver sur les systèmes distribués et les réseaux, 2022) [22], ainsi qu'à la première école d'été sur les environnements distribués et répliqués (DARE 2023) [23].

Structure du Manuscrit

Cette thèse est structurée comme suit :

Première partie : elle explique le contexte de ma recherche, en se concentrant sur les applications cloud et la transition vers l'edge computing.

Dans Chapter 1, j'explique ce que sont les applications cloud, la transition vers l'edge computing et leurs limitations.

Dans Chapter 2, je traite de la transition des systèmes distribués aux systèmes géo-distribués, et je donne une vue générale des solutions existantes avec des exemples pour chacune. J'y souligne également leurs limitations et le besoin d'une approche générique et non intrusive.

Dans Chapter 3, j'expose l'approche existante de Cheops, en expliquant en détail chaque composant. Je discute aussi de leurs limitations et introduis les questions de recherche qui motivent ma thèse.

Deuxième partie : elle décrit les solutions pertinentes existantes qui tentent de répondre à nos questions de recherche, et j'établis un état de l'art basé sur celles-ci.

Dans Chapter 4, j'examine les solutions qui tentent de répondre à la première question de recherche en détail et je présente une comparaison avec notre approche envisagée.

Dans Chapter 5, j'analyse les solutions qui tentent de répondre à la deuxième question de recherche, et je compare ces solutions avec l'approche que nous envisageons.

L'étude concernant la troisième question est présentée avec l'explication de l'approche dans la prochaine partie.

Troisième partie : elle décrit mon approche pour répondre aux trois questions de recherche.

Dans Chapter 6, j'explique la nouvelle architecture de Cheops et la configuration expérimentale que nous utilisons pour valider l'ensemble de nos contributions dans ce

manuscrit.

Dans Chapter 7, je présente ma méthode pour répondre à la première question de recherche, intitulée Cross. Cette méthode permet de répartir n'importe quelle ressource dans une application pour les géo-distribuer à travers des instances.

Dans Chapter 8, je présente mon approche pour répondre à la deuxième question de recherche. J'y explique une méthode pour garantir la cohérence des ressources entre les instances géo-distribuées de toute application, de manière générique et non intrusive.

Dans Chapter 9, j'expose ma solution pour répondre à la troisième question de recherche, en présentant une méthode qui garantit que l'illusion d'une instance unique est maintenue dans Cheops en combinant plusieurs instances individuelles, et en assurant la reproduction de toute opération locale dans cet environnement.

Enfin, je conclus ma thèse avec une conclusion générale et des perspectives de travaux futurs qui pourraient encore améliorer l'approche Cheops.

Bibliography

[1] Edwin Sturrus and Olga Kulikova, „Orchestrating Hybrid Cloud Deployment: An Overview“, *in*: *Computer* 47 (2014), pp. 85–87, URL: `https://api.semanticscholar.org/CorpusID:32351461` (cit. on pp. 12, 21, 179).

[2] Rathinaraja Jeyaraj et al., „Resource Management in Cloud and Cloud-influenced Technologies for Internet of Things Applications“, *in*: *ACM Computing Surveys* 55 (2022), pp. 1–37, URL: `https://api.semanticscholar.org/CorpusID:254735484` (cit. on pp. 12, 21, 179).

[3] Jinke Ren et al., „Collaborative Cloud and Edge Computing for Latency Minimization“, *in*: *IEEE Transactions on Vehicular Technology* 68 (2019), pp. 5031–5044, DOI: `10.1109/TVT.2019.2904244` (cit. on pp. 12, 25, 179).

[4] Mahadev Satyanarayanan et al., „The case for vm-based cloudlets in mobile computing“, *in*: *IEEE pervasive Computing* 8.*4* (2009), pp. 14–23 (cit. on pp. 12, 25, 63, 179).

[5] M. Bukhsh, Saima Abdullah, and Imran Sarwar Bajwa, „A Decentralized Edge Computing Latency-Aware Task Management Method With High Availability for IoT Applications“, *in*: *IEEE Access* 9 (2021), pp. 138994–139008, DOI: `10.1109/ACCESS.2021.3116717` (cit. on pp. 12, 179).

[6] X. Masip-Bruin et al., „Managing the Cloud Continuum: Lessons Learnt from a Real Fog-to-Cloud Deployment“, *in*: *Sensors (Basel, Switzerland)* 21 (2021), DOI: `10.3390/s21092974` (cit. on pp. 12, 26, 179).

[7] Malika Bendechache et al., „Simulating Resource Management across the Cloud-to-Thing Continuum: A Survey and Future Directions“, *in*: *Future Internet* 12 (2020), p. 95, DOI: `10.3390/fi12060095` (cit. on pp. 12, 179).

[8] *Federated Keystone*, `https://docs.openstack.org/keystone/latest/admin/federation/introduction.html`, [Online; accessed 19-Aug-2024] (cit. on pp. 12, 31, 180).

[9] Ying Xiong et al., „Extend cloud to edge with kubeedge“, *in*: *2018 IEEE/ACM Symposium On Edge Computing (SEC)*, IEEE, 2018, pp. 373–377 (cit. on pp. 13, 35, 64, 66, 180).

[10] *Antodote DB*, `https://www.antidotedb.eu/`, [Online; accessed 19-Aug-2024] (cit. on pp. 13, 180).

[11] Ronan-Alexandre Cherrueau, Marie Delavergne, and Adrien Lebre, „Geo-distribute cloud applications at the edge“, *in*: *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*, Springer, 2021, pp. 301–316 (cit. on pp. 13, 29, 33, 34, 38, 40, 41, 180, 181).

[12] Marie Delavergne, „Cheops, a service-mesh to geo-distribute micro-service applications at the Edge“, PhD thesis, Ecole nationale supérieure Mines-Télécom Atlantique, 2023 (cit. on pp. 13, 14, 37, 41, 53, 180–182).

[13] 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*, Springer, 2022, pp. 530–539 (cit. on pp. 13, 16, 34, 41, 146, 181, 184).

[14] Marie Delavergne, Ronan-Alexandre Cherrueau, and Adrien Lebre, „A service mesh for collaboration between geo-distributed services: the replication case“, *in*: *Agile Processes in Software Engineering and Extreme Programming–Workshops: XP 2021 Workshops, Virtual Event, June 14–18, 2021, Revised Selected Papers 22*, Springer International Publishing, 2021, pp. 176–185 (cit. on pp. 13, 34, 41, 100, 181).

[15] Diego Ongaro and John Ousterhout, „In search of an understandable consensus algorithm“, *in*: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014 (cit. on pp. 14, 120, 182).

[16] Rajkumar Buyya, Toni Cortes, and Hai Jin, „Single system image“, *in*: *The International Journal of High Performance Computing Applications* 15.*2* (2001), pp. 124–135 (cit. on pp. 15, 40, 146, 182).

[17] Geo Johns Antony et al., „Thinking out of replication for geo-distributing applications: the sharding case“, *in*: *ICFEC 2024-8th IEEE International Conference on Fog and Edge Computing*, 2024 (cit. on pp. 16, 34, 95, 98, 129, 136, 184).

[18] *Cloud Control Workshop 2023*, https://cloudresearch.org/workshops/, [Online; accessed 19-Aug-2024] (cit. on pp. 16, 184).

[19] *Compas 2022*, https://2022.compas-conference.fr/, [Online; accessed 19-Aug-2024] (cit. on pp. 16, 184).

[20] *Open Infra Summit 2024*, https://www.youtube.com/watch?v=7EZ63DMRJhc, [Online; accessed 19-Aug-2024] (cit. on pp. 17, 184).

[21] *Journées Cloud 2021*, https://journeescloud2021.github.io/, [Online; accessed 19-Aug-2024] (cit. on pp. 17, 185).

[22] *GDR RSD ASF Winter School on Distributed Systems and Networks*, https://sites.google.com/site/rsdwinterschool/home, [Online; accessed 19-Aug-2024] (cit. on pp. 17, 185).

[23] *First Summer School on Distributed and Replicated Environments (DARE 2023)*, https://soft.vub.ac.be/dare23/, [Online; accessed 19-Aug-2024] (cit. on pp. 17, 185).

[24] Yingling Mao, Xiaojun Shang, and Yuanyuan Yang, „Joint Resource Management and Flow Scheduling for SFC Deployment in Hybrid Edge-and-Cloud Network“, *in*: *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications* (2022), pp. 170–179, URL: https://api.semanticscholar.org/CorpusID:249909338 (cit. on p. 21).

[25] Marek Moravcik, Pavel Segec, and Martin Kontsek, „Overview of cloud computing standards“, *in*: *2018 16th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, IEEE, 2018, pp. 395–402 (cit. on p. 21).

[26] Michael D Hogan et al., „NIST-SP 500-291, NIST cloud computing standards roadmap“, *in*: (2011) (cit. on pp. 21, 22).

[27] Swapnil M Parikh, Narendra M Patel, and Harshadkumar B Prajapati, „Resource management in cloud computing: classification and taxonomy“, *in*: *arXiv preprint arXiv:1703.00374* (2017) (cit. on p. 23).

[28] Adriana Mijuskovic et al., „Resource management techniques for cloud/fog and edge computing: An evaluation framework and classification“, *in*: *Sensors* 21.5 (2021), p. 1832 (cit. on p. 23).

[29] Abdullah Yousafzai et al., „Cloud resource allocation schemes: review, taxonomy, and opportunities“, *in*: *Knowledge and information systems* 50 (2017), pp. 347–381 (cit. on p. 23).

[30] Ali Belgacem, „Dynamic resource allocation in cloud computing: analysis and taxonomies“, *in*: *Computing* 104 (2022), pp. 681–710, URL: https://api.semanticscholar.org/CorpusID:246406235 (cit. on p. 23).

[31] Haifeng Lv and Baoming Pu, „Research on the Dynamic Dispatching Algorithm of Cloud Data Center Resource“, *in*: *Proceedings of the 2022 5th International Conference on Electronics, Communications and Control Engineering* (2022), URL: https://api.semanticscholar.org/CorpusID:250360604 (cit. on p. 23).

[32] Debankur Mukherjee et al., „Optimal Service Elasticity in Large-Scale Distributed Systems“, *in*: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1 (2017), pp. 1–28, DOI: 10.1145/3084463 (cit. on p. 24).

[33] Marta Catillo, M. Rak, and Umberto Villano, „Auto-scaling in the Cloud: Current Status and Perspectives“, *in*: (2019), pp. 616–625, DOI: 10.1007/978-3-030-33509-0_58 (cit. on p. 24).

[34] Sam Newman, *Building microservices*, " O'Reilly Media, Inc.", 2021 (cit. on p. 24).

[35] Naghmeh Niknejad et al., „Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation“, *in*: *Inf. Syst.* 91 (2020), p. 101491, DOI: 10.1016/j.is.2020.101491 (cit. on p. 25).

[36] M. Papazoglou and Dimitrios Georgakopoulos, „Introduction: Service-oriented computing“, *in*: *Communications of The ACM* 46 (2003), pp. 24–28, DOI: 10.1145/944217.944233 (cit. on p. 25).

[37] „API Features Individualizing of Web Services: REST and SOAP“, *in*: *International Journal of Innovative Technology and Exploring Engineering* (2019), DOI: 10.35940/ijitee.i1107.0789s19 (cit. on p. 25).

[38] Md Shahedul Alam et al., „A REST and HTTP-based Service Architecture for Industrial Facilities“, *in*: *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)* 1 (2020), pp. 398–401, DOI: 10.1109/ICPS48405.2020.9274792 (cit. on p. 25).

[39] B. Zolfaghari et al., „Content Delivery Networks“, *in*: *ACM Computing Surveys (CSUR)* 53 (2020), pp. 1–34, DOI: 10.1145/3380613 (cit. on pp. 25, 44).

[40] Li Lin et al., „Computation Offloading Toward Edge Computing“, *in*: *Proceedings of the IEEE* 107 (2019), pp. 1584–1607, DOI: 10.1109/JPROC.2019.2922285 (cit. on p. 25).

[41] B. Varghese et al., „Revisiting the Arguments for Edge Computing Research“, *in*: *IEEE Internet Computing* 25 (2021), pp. 36–42, DOI: 10.1109/mic.2021.3093924 (cit. on p. 25).

[42] Dragi Kimovski et al., „Mobility-Aware IoT Application Placement in the Cloud – Edge Continuum“, *in*: *IEEE Transactions on Services Computing* 15 (2022), pp. 3358–3371, DOI: 10.1109/TSC.2021.3094322 (cit. on p. 26).

[43] André Luckow, Kartik Rattan, and S. Jha, „Pilot-Edge: Distributed Resource Management Along the Edge-to-Cloud Continuum“, *in*: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2021), pp. 874–878, DOI: 10.1109/IPDPSW52791.2021.00130 (cit. on p. 26).

[44] E. Carlini et al., „Efficient Resources Distribution for an Ephemeral Cloud/Edge continuum“, *in*: *ArXiv* abs/2107.07195 (2021) (cit. on p. 26).

[45] Bastien Confais, Benoit Parrein, and Adrien Lebre, „Data Location Management Protocol for Object Stores in a Fog Computing Infrastructure“, *in*: *IEEE Transactions on Network and Service Management* 16.*4* (2019), pp. 1624–1637, DOI: 10.1109/TNSM.2019.2929823 (cit. on p. 26).

[46] David W Chadwick et al., „Adding federated identity management to openstack“, *in*: *Journal of Grid Computing* 12 (2014), pp. 3–27 (cit. on p. 31).

[47] *KubeFed*, https://github.com/kubernetes-retired/kubefed, [Online; accessed 19-Aug-2024] (cit. on pp. 32, 35).

[48] Lars Larsson et al., „Decentralized kubernetes federation control plane“, *in*: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 2020, pp. 354–359 (cit. on p. 33).

[49] *Riak*, https://docs.riak.com, [Online; accessed 19-Aug-2024] (cit. on p. 33).

[50] Marc Shapiro et al., „Conflict-free replicated data types“, *in*: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, Springer, 2011, pp. 386–400 (cit. on pp. 33, 119, 121).

[51] Enrique Saurez et al., „OneEdge: An Efficient Control Plane for Geo-Distributed Infrastructures“, *in*: *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’21, Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 182–196, ISBN: 9781450386388, DOI: `10.1145/3472883.3487008`, URL: `https://doi.org/10.1145/3472883.3487008` (cit. on pp. 35, 59, 66).

[52] James C Corbett et al., „Spanner: Google globally distributed database“, *in*: *ACM Transactions on Computer Systems (TOCS)* 31.*3* (2013), pp. 1–22 (cit. on p. 44).

[53] *Amazon S3 Bucket Object store*, `https://aws.amazon.com/s3/`, [Online; accessed 19-Aug-2024] (cit. on p. 44).

[54] *Couchbase*, `https://www.couchbase.com/`, [Online; accessed 19-Aug-2024] (cit. on p. 44).

[55] Pengzhan Hao et al., „Edgecourier: an edge-hosted personal service for low-bandwidth document synchronization in mobile cloud storage services“, *in*: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–14 (cit. on p. 45).

[56] Bettina Kemme, Andrea Bartoli, and Ozalp Babaoglu, „Online reconfiguration in replicated databases based on group communication“, *in*: *Proceedings of the International Conference on Dependable Systems and Networks*, 2020, pp. 117–130 (cit. on p. 45).

[57] Lara Lorna Jiménez and O. Schelen, „HYDRA: Decentralized Location-Aware Orchestration of Containerized Applications“, *in*: *IEEE Transactions on Cloud Computing* 10 (2022), pp. 2664–2678, DOI: `10.1109/TCC.2020.3041465` (cit. on p. 53).

[58] Jan Kalbantner et al., „P2PEdge: A Decentralised, Scalable P2P Architecture for Energy Trading in Real-Time“, *in*: *Energies* (2021), DOI: `10.3390/EN14030606` (cit. on p. 53).

[59] Lequn Chen et al., „Symphony: Optimized Model Serving using Centralized Orchestration“, *in*: *ArXiv* abs/2308.07470 (2023), DOI: `10.48550/arXiv.2308.07470` (cit. on p. 53).

[60] Breno G. S. Costa et al., „Orchestration in Fog Computing: A Comprehensive Survey“, *in*: *ACM Computing Surveys (CSUR)* 55 (2022), pp. 1–34, DOI: `10.1145/3486221` (cit. on p. 53).

[61] Lara Lorna Jimenez and Olov Schelen, „HYDRA: Decentralized Location-Aware Orchestration of Containerized Applications“, *in*: *IEEE Transactions on Cloud Computing* 10.*4* (2022), pp. 2664–2678, DOI: 10.1109/TCC.2020.3041465 (cit. on pp. 56, 66).

[62] Petar Maymounkov and David Mazieres, „Kademlia: A peer-to-peer information system based on the xor metric“, *in*: *International workshop on peer-to-peer systems*, Springer, 2002, pp. 53–65 (cit. on p. 56).

[63] *liqo*, https://liqo.io, [Online; accessed 19-Aug-2024] (cit. on pp. 57, 58, 66, 146).

[64] *Virtual Kubelet*, https://github.com/virtual-kubelet/virtual-kubelet, [Online; accessed 19-Aug-2024] (cit. on p. 58).

[65] Nan Wang et al., „ENORM: A framework for edge node resource management“, *in*: *IEEE transactions on services computing* 13.*6* (2017), pp. 1086–1099 (cit. on pp. 60, 66).

[66] Sangmin Lee et al., „Shard manager: A generic shard management framework for geo-distributed applications“, *in*: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 553–569 (cit. on pp. 61, 66, 95, 98, 99).

[67] Joao Monteiro Soares et al., „Redesigning Cloud Platforms for Massive Scale Using a P2P Architecture“, *in*: *2017 IEEE International Conference on Cloud Computing Technology and Science CloudCom*, 2017, pp. 57–64, DOI: 10.1109/CloudCom.2017.17 (cit. on pp. 62, 66).

[68] Genc Tato et al., „Sharelatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application“, *in*: *Proceedings of the 3rd Workshop on Middleware for Edge Clouds and Cloudlets*, 2018, pp. 8–15 (cit. on pp. 65, 66, 116).

[69] Andrew Jeffery, Heidi Howard, and Richard Mortier, „Rearchitecting kubernetes for the edge“, *in*: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, 2021, pp. 7–12 (cit. on pp. 71, 80).

[70] Enrique Saurez et al., „A drop-in middleware for serializable DB clustering across geo-distributed sites“, *in*: *Proc. VLDB Endow.* 13.*12* (Aug. 2020), pp. 3340–3353, ISSN: 2150-8097, DOI: 10.14778/3415478.3415555, URL: https://doi.org/10.14778/3415478.3415555 (cit. on pp. 73, 80).

[71] Tobias Pfandzelter et al., „Managing data replication and distribution in the fog with fred“, *in*: *Software: Practice and Experience* 53.*10* (2023), pp. 1958–1981 (cit. on pp. 74, 80).

[72] Seyed Hossein Mortazavi et al., „Sessionstore: A session-aware datastore for the edge“, *in*: *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, IEEE, 2020, pp. 59–68 (cit. on pp. 76, 80).

[73] Manuel Bravo et al., „{UniStore}: A fault-tolerant marriage of causal and strong consistency“, *in*: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 923–937 (cit. on pp. 77, 80).

[74] M Perrin et al., „State-machine replication for planet-scale systems“, *in*: (2020) (cit. on pp. 79, 80).

[75] Anthony Kougkas et al., „Chronolog: a distributed shared tiered log store with time-based data ordering“, *in*: *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST 2020)*, 2020 (cit. on p. 80).

[76] Eric Brewer, „CAP twelve years later: How the" rules" have changed“, *in*: *Computer* 45.*2* (2012), pp. 23–29 (cit. on p. 82).

[77] *Rabbit MQ AMQP*, https://www.rabbitmq.com/tutorials/amqp-concepts, [Online; accessed 19-Aug-2024] (cit. on p. 87).

[78] *ArangoDB*, https://arangodb.com/, [Online; accessed 19-Aug-2024] (cit. on p. 87).

[79] *HA Proxy*, https://www.haproxy.org/, [Online; accessed 19-Aug-2024] (cit. on p. 87).

[80] *CouchDB*, https://couchdb.apache.org/, [Online; accessed 19-Aug-2024] (cit. on p. 91).

[81] Daniel Balouek et al., „Adding Virtualization Capabilities to the Grid'5000 Testbed“, *in*: *Cloud Computing and Services Science*, ed. by Ivan I. Ivanov et al., vol. 367, Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20, ISBN: 978-3-319-04518-4, DOI: `10.1007/978-3-319-04519-1_1` (cit. on p. 92).

[82] *STACK gitlab*, `https://gitlab.inria.fr/discovery/cheops`, [Online; accessed 19-Aug-2024] (cit. on p. 92).

[83] Michał Król et al., „Shard scheduler: Object placement and migration in sharded account-based blockchains“, *in*: *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, 2021, pp. 43–56 (cit. on pp. 96, 98, 99).

[84] *MySQL*, `https://www.mysql.com/`, [Online; accessed 19-Aug-2024] (cit. on p. 96).

[85] *MongoDB*, `https://www.mongodb.com/docs/`, [Online; accessed 19-Aug-2024] (cit. on p. 96).

[86] *Elasticsearch*, `https://www.elastic.co/elasticsearch`, [Online; accessed 19-Aug-2024] (cit. on p. 96).

[87] *Google Bigtable*, `https://cloud.google.com/bigtable`, [Online; accessed 19-Aug-2024] (cit. on p. 96).

[88] Pakorn Kookarinrat and Yaowadee Temtanapat, „Analysis of range-based key properties for sharded cluster of mongodb“, *in*: *2015 2nd International Conference on Information Science and Security (ICISS)*, IEEE, 2015, pp. 1–4 (cit. on p. 99).

[89] Daan Leijen, Benjamin Zorn, and Leonardo de Moura, „Mimalloc: Free list sharding in action“, *in*: *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*, Springer, 2019, pp. 244–265 (cit. on p. 99).

[90] Caio H Costa, PHM Maia, F Carlos, et al., „Sharding by hash partitioning“, *in*: *Proceedings of the 17th International Conference on Enterprise Information Systems*, vol. 1, 2015, pp. 313–320 (cit. on p. 99).

[91] Lin Xiao et al., „ShardFS vs. IndexFS: Replication vs. caching strategies for distributed metadata management in cloud storage systems“, *in*: *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 236–249 (cit. on p. 99).

[92] Tim Rentsch, „Object oriented programming“, *in*: *ACM Sigplan Notices* 17.*9* (1982), pp. 51–57 (cit. on p. 101).

[93] Victor Yodaiken, „Understanding Paxos and other distributed consensus algorithms“, *in*: *ArXiv* abs/2202.06348 (2022), URL: `https://api.semanticscholar.org/CorpusID:246823109` (cit. on p. 119).

[94] Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu, „Generalizing operational transformation to the standard general markup language“, *in*: *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, 2002, pp. 58–67 (cit. on p. 120).

[95] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker, „Pure operation-based replicated data types“, *in*: *arXiv preprint arXiv:1710.04469* (2017) (cit. on p. 121).

[96] Peter Bailis and Ali Ghodsi, „Eventual consistency today: Limitations, extensions, and beyond“, *in*: *Communications of the ACM* 56.*5* (2013), pp. 55–63 (cit. on p. 121).

[97] Kenneth Birman, Andre Schiper, and Pat Stephenson, „Lightweight causal and atomic group multicast“, *in*: *ACM Transactions on Computer Systems (TOCS)* 9.*3* (1991), pp. 272–314 (cit. on pp. 121, 124).

[98] Alexey Gotsman et al., „'Cause I'm strong enough: Reasoning about consistency choices in distributed systems“, *in*: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 371–384 (cit. on p. 121).

[99] Maher Suleiman, Michele Cart, and Jean Ferrié, „Concurrent operations in a distributed and mobile collaborative environment“, *in*: *Proceedings 14th International Conference on Data Engineering*, IEEE, 1998, pp. 36–45 (cit. on p. 122).

[100] Leslie Lamport, „Time, clocks, and the ordering of events in a distributed system“, *in*: *Commun. ACM* 21.*7* (July 1978), pp. 558–565, ISSN: 0001-0782, DOI: `10.1145/359545.359563`, URL: `https://doi.org/10.1145/359545.359563` (cit. on p. 124).

[101] Leslie Lamport, „The part-time parliament“, *in*: *ACM Transactions on Computer Systems (TOCS)* 16.*2* (1998), pp. 133–169 (cit. on pp. 124, 130, 131).

[102] Stéphane Weiss, Pascal Urso, and Pascal Molli, „Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks“, *in*: *2009 29th IEEE International Conference on Distributed Computing Systems*, IEEE, 2009, pp. 404–412 (cit. on pp. 129, 133).

[103] Geoffrey Litt et al., „Peritext: A crdt for collaborative rich text editing“, *in*: *Proceedings of the ACM on Human-Computer Interaction* 6.*CSCW2* (2022), pp. 1–36 (cit. on pp. 129, 130).

[104] Matthew Weidner, Heather Miller, and Christopher Meiklejohn, „Composing and decomposing op-based CRDTs with semidirect products“, *in*: *Proc. ACM Program. Lang.* 4.*ICFP* (Aug. 2020), DOI: `10.1145/3408976`, URL: `https://doi.org/10.1145/3408976` (cit. on pp. 129, 133).

[105] Martin Kleppmann and Alastair R Beresford, „Automerge: Real-time data sync between edge devices“, *in*: *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk. org/abstract/S4-P5-Kleppmann-Automerge. pdf*, sn, 2018, pp. 101–105 (cit. on pp. 129, 133).

[106] *Go-lang*, `https://go.dev/doc/`, [Online; accessed 19-Aug-2024] (cit. on p. 129).

[107] *Git Hooks*, `https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`, [Online; accessed 19-Aug-2024] (cit. on p. 136).

[108] P. Novotný, A. Wolf, and B. J. Ko, „Discovering service dependencies in mobile ad hoc networks“, *in*: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)* (2013), pp. 527–533 (cit. on p. 146).

[109] Y. Woldeyohannes and Yuming Jiang, „Measures for Network Structural Dependency Analysis“, *in*: *IEEE Communications Letters* 22 (2018), pp. 2052–2055, DOI: `10.1109/LCOMM.2018.2864109` (cit. on p. 146).

[110] M. Göring and A. Fay, „Modeling change and structural dependencies of automation systems“, *in*: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies and Factory Automation (ETFA 2012)* (2012), pp. 1–8, DOI: `10.1109/ETFA.2012.6489540` (cit. on p. 146).

[111] Hinde Lilia Bouziane, Christian Pérez, and Thierry Priol, „Extending software component models with the master–worker paradigm“, *in*: *Parallel Computing* 36.*2-3* (2010), pp. 86–103 (cit. on p. 149).

[112] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra, „Modeling and analyzing MAPE-K feedback loops for self-adaptation", *in*: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, IEEE, 2015, pp. 13–23 (cit. on p. 174).

[113] Gregor Kiczales et al., „Aspect-oriented programming", *in*: *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*, Springer, 1997, pp. 220–242 (cit. on p. 178).

[114] Renaud Pawlak et al., „JAC: A flexible solution for aspect-oriented programming in Java", *in*: *Metalevel Architectures and Separation of Crosscutting Concerns: Third International Conference, REFLECTION 2001 Kyoto, Japan, September 25–28, 2001 Proceedings 3*, Springer, 2001, pp. 1–24 (cit. on p. 178).

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Cheops reloaded : Nouvelles avancées dans le découplage de la géo-distribution de la logique métier des applications

Mot clés : Informatique nuage, Informatique périphérie, Découplage application, Orchestrateur

Résumé : La transition du cloud computing centralisé vers des applications géo-distribuées est essentielle pour répondre aux exigences modernes de services à faible latence, haute disponibilité et résilience. Cependant, les solutions existantes de géo-distribution nécessitent souvent des modifications intrusives du code de l'application. Ma thèse étend Cheops, un middleware qui découple la géo-distribution de la logique applicative, offrant une solution non intrusive et générique pour déployer une application sur des instances géographiquement distribuées. S'appuyant sur les principes de Cheops, "local-first" et "collaborative-then", mes recherches introduisent Cross, un mécanisme de collaboration par fragmentation pour partitionner les ressources entre différents sites, ainsi qu'une nouvelle approche pour découpler la gestion de la cohérence de la logique applicative, garantissant la synchronisation entre les instances. De plus, la gestion des dépendances assure que les opérations effectuées sur une instance sont reproductibles à travers les instances géo-distribuées, maintenant l'illusion d'une application unifiée et localisée sur un seul site. Cheops utilise Scope-lang, un langage spécifique au domaine (DSL), pour faciliter cela sans modifier la logique de l'application. Cette extension de Cheops renforce davantage la séparation entre la géo-distribution et la logique métier de l'application.

Title: Cheops reloaded : Further steps in Decoupling Geo-Distribution from Application business logic

Keywords: Cloud computing, Edge Computing, Decoupling application, Orchestrator

Abstract: The shift from centralized cloud computing to geo-distributed applications is critical for meeting modern demands for low-latency, highly available, and resilient services. However, existing geo-distribution solutions often require intrusive modifications to application code. My thesis extends the Cheops framework, a middleware that decouples geo-distribution from application logic, offering a non-intrusive and generic solution for deploying an application across geographically distributed instances. Building on the Cheops principles of "local-first" and "collaborative-then," my research introduces Cross, a shard collaboration mechanism for partitioning resources across sites, and a new approach to decoupling consistency from the application logic, ensuring synchronization between instances. Additionally, dependency management guarantees that operations performed on one instance are reproducible across geo-distributed instances, maintaining the illusion of a unified, single-site application. Cheops uses Scope-lang, a Domain-Specific Language (DSL), to facilitate this without altering application logic. This extension of Cheops, further enhances the separation of geo-distribution from the application business logic.