CONTROL PLANE FOR SITUATION-AWARENESS APPLICATIONS ON GEO-DISTRIBUTED RESOURCES

A Dissertation
Presented to
The Academic Faculty

By

Enrique Saurez

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

May 2022

© Enrique Saurez 2022

CONTROL PLANE FOR SITUATION-AWARENESS APPLICATIONS ON GEO-DISTRIBUTED RESOURCES

Thesis committee:

Professor Umakishore Ramachandran, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Mostafa Ammar
School of Computer Science
Georgia Institute of Technology

Professor Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Alexandros Daglis
School of Computer Science
Georgia Institute of Technology

Dr. Bharath Balasubramanian
Senior Software Engineer
Google

Date approved: April 28, 2021

To my wife, Laura,
my parents, Marcial and Ana,
and my grandfather, Marcial,
for all their love and encouragement.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Umakishore Ramachandran, for his relentless support and motivation during my academic path. During my Ph.D., Professor Kishore gave me immense freedom to explore different projects and internships, which helped me grow as a researcher. I am grateful for his always cheerful encouragement for me to aim higher.

I want to thank the committee members, Professor Mostafa Ammar and Professor Ada Gavrilovska, for taking their valuable time to be on my committee and providing feedback that greatly improved this thesis. Next, I would like to thank Professor Alexandros Daglis for his guidance during the OneEdge project; his expertise was fundamental to completing the project. Finally, I am especially thankful to Dr. Bharath Balasubramanian for his mentoring over multiple years during our collaborations and his invaluable guidance at different stages of my Ph.D.

A special thanks to Harshit Gupta and Adam Hall, who did not only collaborate with me on many of my projects, but I am also greatly in debt for their help and friendship. I would also like to thank my colleagues at the EPL, Wonhee Cho, Ashish Bijlani, Zhuangdi (Andy) Xu, Alan Nussbaum, Manasvini Sethuraman, Tyler Landle, Anirudh Sarma, Jinsun Yoo, and Difei Cao. I also want to thank our special guests from Stuttgart, Prof. Kurt Rothermel, Ruben Mayer, Henriette Röger, and Sukanya Bhowmik, for the always fun discussions and projects we worked on together. I am also grateful to my external collaborators, Richard Schlichting, Shankaranarayanan Puzhavakath Narayanan, and Zhe Huang.

Personally, I want to thank my parents and grandfather for their love and support in pursuing my goals. I am forever indebted to Laura, my wife, for her patience, love, and inspiration throughout my years at Georgia Tech; she was always there for me and supported me through all my obstacles. Last but not least, I am thankful to my friends, David and Maria José, for their support throughout this stage of my life.

TABLE OF CONTENTS

Acknowledgments iv

List of Tables xii

List of Figures xiii

List of Acronyms xviii

Summary xix

Chapter 1: Introduction 1

1.1 Problem Statement 3

1.2 Thesis Statement 5

1.3 Contributions 5

1.4 Roadmap 7

Chapter 2: Background 8

2.1 Geo-distributed Resources 8

2.2 Situation-awareness applications 8

2.3 Programming models for situation-awareness applications 9

Chapter 3: Programming model 11

3.1 Situation-awareness applications 11
3.1.1 Types of applications 12
3.2 Towards a programming model 15
3.3 Programming model 18
3.4 Application requirements 19
3.4.1 End-to-end latency service-level objective (SLO) 19
3.4.2 Bandwidth 22
3.4.3 Spatial Affinity 22
3.5 API and runtime handlers 24
3.5.1 Logical partition of applications 25
3.5.2 Inter-component communication 25
3.5.3 Data Management 27
3.6 Example of an application implementation 29
3.6.1 Configuration file 29
3.6.2 Code representation 30
3.7 Effect on application implementation 32
3.8 Effect of programming model on the control plane 33
3.8.1 Multiple data-flow graph instances per application and sharing . . . 34
3.8.2 Migration API 35
3.9 Limitations 35
3.10 Conclusion 37

Chapter 4: Architecture for a control plane for geo-distributed resources 38

4.1 Application life cycle overview 38
4.2 Requirements 39
4.3 Challenges 41
4.4 Related Work 42
4.5 Overview of the components in the control plane 43
4.6 Scheduler 45
4.7 Monitoring and Policy Definition 48
4.8 Control Plane Managers and Runtime Library 49
4.9 State Manager 51
4.10 Discussion of the architecture and requirements 52
4.11 Distribution of control plane components 53
4.11.1 Preview of following chapters 54

Chapter 5: Analysis of a centralized architecture and its limitations 55

5.1 Background: Kubernetes—a centralized control plane 55
5.1.1 Control plane architecture 55
5.1.2 State in etcd 58
5.1.3 Kubernetes API and workflow 59
5.1.4 Scheduling 61
5.2 Designing a geo-distributed control plane with Kubernetes for situation-awareness applications 63
5.2.1 Enhancing Kubernetes to support spatial and end-to-end deployments 64
5.2.2 Distribution of components in a centralized architecture and workflow 65

5.3 Limitations of a centralized design for situation-awareness applications and geo-distributed infrastructure 67

5.3.1 Geo-distributed Kubernetes extensions 69

5.3.2 Discussion of Kubernetes limitations 71

5.4 Chapter summary 72

Chapter 6: Decentralized architecture 74

6.1 Architecture overview and distribution 74

6.2 Workflow 76

6.3 Local deployments and peer-to-peer coordination 77

6.3.1 Discovery and deployment protocol 77

6.3.2 Join protocol 79

6.4 Migrations 81

6.4.1 QoS-driven migration 82

6.4.2 Application state management 85

6.4.3 Peer-to-peer coordination 88

6.5 Dynamic resource reallocation: workload-driven migration 88

6.6 Implementation 90

6.7 Evaluations 91

6.7.1 Platform 91

6.7.2 Starting the Foglets system and application components 92

6.7.3 Microbenchmarks 93

6.7.4 Dynamic workload-driven migration 97

6.7.5 Proactive migration 98

6.8 Chapter summary and limitations 98

Chapter 7: Hybrid architecture 100

7.1 Insights and benefits of hybrid 100

7.2 Architecture overview 102

7.3 Workflow 104

7.3.1 Local-domain overview 104

7.3.2 Global-domain overview 104

7.4 Multi micro-datacenter mechanisms 106

7.4.1 Deflection 106

7.4.2 Scheduling 107

7.5 Reactive policies 112

7.5.1 Hierarchical Monitoring 113

7.5.2 Dynamic resource allocation 114

7.6 Deployment and multi-domain coordination 115

7.6.1 Re-execution of requests after aborts 116

7.7 Performance optimizations 117

7.7.1 Enhanced two-phase commit 118

7.7.2 Transaction pipelining 120

7.8 Fault Tolerance 122

7.9 Implementation 122

7.10 Evaluations 123

7.10.1 Experimental platform 123

7.10.2 Microbenchmarks 124

7.10.3 End-to-end evaluations 132

7.10.4 Discussion: extending OneEdge evaluations to higher request rates and scalability limitations 137

7.11 Chapter summary 138

Chapter 8: Related work 140

8.1 Programming models for situation-awareness 140

8.2 Control plane architectures and mechanisms 142

8.3 Scheduling algorithms 143

8.4 Monitoring 144

8.5 Dynamic reconfigurations 145

8.6 Application migration 147

Chapter 9: Discussion and lessons learned 148

9.1 Control plane design for situation awareness application 148

9.2 Control design for geo-distributed infrastructure 149

9.3 Lessons learned 151

9.3.1 Leverage application semantics and infrastructure knowledge 151

9.3.2 Focus on the real objective 152

Chapter 10: Conclusion and future directions 153

10.1 Conclusion 153

10.2 Future Directions 155

10.2.1 Control plane design for geo-distributed resources 156

10.2.2 Control plane design for situation-awareness applications 159

Appendices 162

Appendix A: Pseudocode for connected cars application 163

References 168

LIST OF TABLES

3.1 Programming model API: communication primitives. 26

3.2 Programming model API: manipulating the local object store. 27

3.3 Programming model handlers: invoked by the the runtime on message arrival. 28

6.1 Startup times for different configurations of Docker images 92

7.1 Summary of parameters for microbenchmarks of OneEdge. 124

LIST OF FIGURES

2.1 Situation-awareness application example and dataflow graph: police officer finding a missing child 9

3.1 An exemplar of situation awareness applications – Connected Vehicles, a coordinated application. Cars in the same spatial locale have their individual views fused by the sub-regional view; the regional view fuses sub-regional views of adjacent spatial locales. The sub-regional view also sends feedback to the cars in a latency-sensitive manner. 13

3.2 The standalone application process individual drones to calculate their pose based on the sensor inputs (camera and inertial measurement unit). The pose information is then feedback to the drone to continue its navigation. . . 15

3.3 The connected car application is annotated with three different types of service-level objectives. The sub-regional views have two colors associated with different areas of interest, where two cars are in the yellow region and one car is in the blue region; corresponding instances manage each of the spatial regions. The $S_{i,j}$ presents the end-to-end latency requirements of the corresponding j stage coming from node i. The $D_{i,j}$ presents the bandwidth requirements for the link going from node i to node j. 20

3.4 A generic pipeline that explains the different components of the tolerable latency staleness. $S_{(i-1,i)}$ is the acceptable latency starting at the output of the client to the input of the node i. It is composed of both computational latency C_j and transmission latency $T_{j-1,j}$. 21

3.5 Connected cars dataflow graph. The dataflow graph has three stages: db-scan, prune, and concatenation. The first two stages are part of the *sub-region map fusion*, and the last one is part of the region map component. The last sub-region map fusion component also sends data back to the client. This application has two service-level objectives: a latency staleness bound $S_{(2,0)}$ of 100 ms and spatial affinity constraints for all three nodes, with different areas-of-interest for the third node than the first two nodes. 28

4.1 The control plane is composed of four main components: managers, scheduler, monitoring and policy, and state manager. The managers are further split into three main components: control plane manager, local manager, and runtime library. These logical components perform all the operations required by the control plane for managing geo-distributed resources and situation-awareness applications. 44

4.2 Relationship between functional requirements and logical components of the control plane architecture. The components highlighted in gray (left) focus on both control requirements **R1** and **R2**. The components highlighted in blue (right) are the main drivers behind providing dynamicity and supporting the situation-awareness application's special requirements. . 52

5.1 Architecture of Kubernetes. It comprises two main components: the *Control Plane* and the *Worker Nodes*. There is one logical control plane for the overall Kubernetes system and one instance of *Worker Nodes* for each server. 56

5.2 Three stages of the lifecycle of a *pod* creation in Kubernetes. First, the client submits the request to the API server, and the request is written to etcd. Then, in the second stage, a *watch* is triggered, the scheduler finds a suitable worker to run the pod, and the selection is saved to etcd. Next, in the third stage, the API server triggers the corresponding Kubelet, which deploys the application locally in the server using the container runtime. Finally, the result is saved to etcd when completed successfully (brown arrow). 60

5.3 Extension of Kubernetes to support spatial and end-to-end latency resource scheduling and reconfiguration. An additional application controller is added to handle the semantics of these two deployment requirements, as well as to support atomic dataflow-graph deployments. 64

5.4 Workflow for application deployment in a micro-datacenter using Kubernetes. 66

5.5 Experimental evaluation of an extended Kubernetes to support spatial and end-to-end (E2E) latency requirements. The latency breakdown includes container cold start. 67

5.6 Experimental evaluation of extended Kubernetes to support spatial and E2E latency requirements. Latency breakdown when using a warm container. . . 68

6.1 Foglets architecture. Foglets comprises four components: the registry service, the discovery service, the *local manager*, and the worker process. The registry service and discovery service have geo-distributed instances. There is one *local manager* per micro-datacenter (µDC). Each server can run multiple worker processes, one per application component of potentially different applications. 75

6.2 Discovery and Deployment Protocol 78

6.3 Join Protocol 80

6.4 Join Protocol: the Discovery service gives a list of µDCs to the requesting client node. The protocol pictorially shown above results in the client choosing a parent to join from the list. 81

6.5 Quality-of-Service migration. Foglets migration moves latency-sensitive components to a more suitable micro-datacenter if the latency exceeds the threshold (*i.e.*, $\alpha \cdot T$ for proactive migration). Initially, in step 1, the blue car appears, and it is deployed across micro-datacenter *A* and *B*. At around the same time, in step 2, the brown car appears and gets assigned to micro-datacenter *C* and *B*, sharing some of the components. Finally, in step 3, the blue car moves away, causing the latency to increase. Consequently, Foglets proactively migrates the first component in micro-datacenter *A* to the already deployed one in *B*. 83

6.6 State Migration. The migration happens from the newest data towards the oldest data. Each step migrates a *chunk* of size *M*. Once the data is migrated, the data range is updated to show that the new instance in μDC C is the current owner of the data. If data for the old range [T_0,T_n] is requested, the data would need to be fetched from the old instance in μDC B. 86

6.7 Comparison of Discovery-Join and Discovery-Deployment Operations. Error bars represent the 25% and 75% percentiles. 94

6.8 Proactive Migration Operation. As a point of comparison, we show the network round-trip time. Error bars represent the 25% and 75% percentiles. 96

6.9 Workload Driven Migration. Over time μDC *2* accepts more clients to offload the work from μDC *1*. 96

6.10 QoS-driven Proactive Migration. Over time the clients are migrated to the micro-datacenter that is geo-local to the clients moving in different directions. 97

7.1 OneEdge's System Architecture. The global domain manager (left) coordinates with all the local domain managers in each µDC (right blow-up). Additionally, there are three more components in the in the local domain: monitoring subsystem, runtime library, and container runtime. 102

7.2 The global manager comprises five components: monitoring manager, request queue, resource scheduler, aggregate state, and transaction manager. Requests are added to the request queue by the local domains, monitoring, and transaction managers. The resource scheduler then processes the requests, and the transaction manager finally executes them. 105

7.3 The *transaction manager* is split into two subcomponents: *pending commands* and *command executor*. The *pending commands* maintains a directed-acyclic graph with the resource management actions (i.e., transactions) defined by the scheduler until the *command executor* completes them. The *command executor* is the entity in charge of coordinating with each of the associated local managers for each transaction. 121

7.4 Impact of Pipelining Optimization on Aggregate Throughput. 126

7.5 MCF of baseline and enhanced two-phase commit (2PC) for constrained resources at a µDC and typical coordinated and standalone application request rates from the SF cabs dataset (Table 7.1). The blow-up shows the increase in MCF and the µDC's remaining available resources for enhanced 2PC at higher request rates. 127

7.6 Standalone deployment latency: comparison between centralized and OneEdge's control planes. 129

7.7 Allocation imbalance for standalone application request handling at proximal µDC vs. at the *global manager*. Shown results are for the cell emulated in the West US Azure region. 130

7.8 Deployment latency for standalone application request handling at proximal µDC vs. at the *global manager*. Shown results are for the cell emulated in the West US Azure region. 131

7.9 End-to-end Evaluation of Hybrid architecture: Spatial alignment for the coordinated application: OneEdge vs. greedy placement. 134

7.10 End-to-end evaluation of the hybrid architecture: deployment latency for standalone applications. 136

7.11 End-to-end evaluation of hybrid architecture: end-to-end latency SLO violations detected for the coordinated application. 136

A.1 Connected cars—Json application configuration. 163

A.2 Connected cars—Bootstrap function. 164

A.3 Connected cars—Prune application component. 164

A.4 Connected cars—Dbscan application component interface. 165

A.5 Connected cars—Dbscan application component: auxiliary functions. . . . 165

A.6 Connected cars—Dbscan application component: “on send up”. 166

A.7 Connected cars—Dbscan application component: “on migration end”. . . . 166

A.8 Connected cars—Spatial affinity function. 167

LIST OF ACRONYMS

μDC micro-datacenter

2PC two-phase commit

AoI area of interest

API application programming interface

AR augmented reality

DC datacenter

DFG dataflow graph

E2E end-to-end

IoT internet of things

MCF mean conflict fraction

QoS quality-of-service

RRP resource requirement profile

RTT round-trip time

SLA service-level agreement

SLO service-level objective

VM virtual machine

WAN wide-area network

SUMMARY

Situation-awareness applications generate actionable knowledge from sensor and user data. Two trends are unlocking new situation-awareness applications: geo-distributed resources and pervasive sensors. Geo-distributed computing infrastructure is now available worldwide, ranging from multi-region cloud deployments to newer 5G edge deployments. On the other hand, pervasive sensors have seen a boom, with examples like geo-distributed camera deployments, smart cities, and users' gadgets (smartphones). Having computational resources closer to the data source improves the response time and the efficient use of the available resources. Geo-distributed resources reduce the physical distance to the data source, cutting the time it takes to transmit, filter, and process information, reducing unnecessary data transmission. However, efficient management of resources is challenging for densely geo-distributed resources while also providing spatio-temporal context and latency quality-of-service objectives.

This dissertation proposes a control plane that makes three contributions for efficiently managing situation awareness applications running on geo-distributed computational resources:

1. It defines a programming model capable of expressing situation-awareness applications and their requirements. Additionally, it defines a new taxonomy for situation-awareness applications.

2. It describes the requirements of the control plane and the components needed to support geo-distributed resources and situation-awareness applications.

3. It proposes an efficient control plane architecture and mechanisms to support the above requirements.

The first contribution of this work extends the data-flow graph programming model to better suit the geo-distributed context of both applications and computational resources.

Then, we analyze and classify situation-awareness applications that benefit from geo-distributed resources. Finally, it presents ways to represent the requirements of such applications as part of an extended data-flow graph.

The second contribution of this thesis defines the different building blocks required to manage geo-distributed resources efficiently. Then, starting from the programming model, we show how the different components interact with the geo-distributed physical computational resources. Finally, we analyze why state-of-the-art centralized control planes cannot fulfill situation-awareness application's requirements.

Next, this dissertation contributes Foglets, a fully decentralized control plane architecture. A fully decentralized design reduces the overhead of a centralized design while still providing low-latency access to computational resources. In addition, we present mechanisms to propagate information to allow entirely local decisions and support application migrations between different computational resources' localities.

Finally, this thesis proposes OneEdge, a hybrid control plane architecture, extending the decentralized and centralized designs from the previous contributions. The control plane for geo-distributed resources has an inherent trade-off between response time and the quality of decisions. The main factor that defines the trade-off is the transmission latency incurred between the resources being managed and the control plane component making those decisions for the resources. For example, if the control plane decisions are made in a centralized fashion, we can calculate optimal decisions, but there is a high likelihood that all these decisions would incur at least one wide-area network round-trip, given that the resources are geo-distributed. On the other hand, the latency can be minimized if the control plane components are close to the resources, but then the complexity and cost of maintaining an up-to-date view of other resources and making optimal decisions are increased. We present an architecture that allows us to configure the control plane for different trade-off points and provide all the required service-level objectives defined in the programming model.

CHAPTER 1
INTRODUCTION

More cloud services and applications are being deployed across multiple geographical regions, making them geo-distributed. This transition to a more densely geo-distributed infrastructure usage is partly due to the need for faster response time, higher availability, and more stringent policy compliance (i.e., GDPR [1]). Current examples of this change are multi-region cloud deployments [2] and computational resources colocated with wireless infrastructure–like LTE or 5G deployments [3]. Meanwhile, a newer generation of sensor technology is shaping how this infrastructure is built; these sensors include augmented reality (AR), widespread connected camera deployments, and the internet of things (IoT) sensors. In the following, we list some requirements of exemplar applications:

- Augmented reality overlays information to the user and requires processing with bounded E2E low latency to be useful. Otherwise, any information would be out-of-sync with the user's point of view.
- IoT sensors generate massive amounts of data and demand intelligent computation placement to avoid overburdening the network infrastructure.
- Video streams from camera deployments generate sensitive information and must be pre-processed near the camera for privacy reasons and due to bandwidth limitations, as one 4K camera can generate up to 20 Mbps of data when encoded using HEVC [4].
- The nearby objects detected by autonomous cars in an intersection need to be processed by the same application instance such that the information can be merged to improve the overall cars detection accuracy and avoid unnecessary accidents.

Situation-awareness applications compute actionable insights from sensors and user data to improve people's lives. This application category has the most to gain from the two trends presented before, novel sensor technologies and geo-distributed datacenters. New sensor technologies will allow more detailed information to be gathered (i.e., cameras) and more satisfying feedback to the user (i.e., augmented reality). On the other hand, geo-distributed resources allow lower latency and higher throughput access to computational resources, allowing tighter control loops and faster interactions with the world.

Despite the benefits for situation-awareness applications, managing densely geo-distributed datacenters introduces various technical problems that could hinder their use for this type of application. In a data center environment, available computational resources are managed by an entity known as the control plane. State-of-the-art control plane implementations work well within a traditional datacenter since their design has evolved around applications running in the Cloud. Cloud environments use network topologies, like Clos [5], that are homogeneous with multiple paths between the same endpoints, all with similar and low communication latencies. In turn, these networks connect 50 thousand or more servers [6]. However, geo-distributed datacenters (*i.e.*, micro-datacenter) differ in two main aspects. First, each of these micro-datacenters would have a smaller footprint [7] than regular data centers due to cost. For example, in current China edge deployments [8], each micro-datacenter has between 2 to 43 servers, with up to seven such micro-datacenters in a city. Resource management of geo-distributed infrastructures is challenging due to this relative per-site scarcity of computational resources and the inherent properties of situation awareness applications (*e.g.*, workload surges and high mobility of clients). Second, the geo-distributed micro-datacenters will not have such homogenous connectivity between each other. State-of-the-art architectures were created with a different trade-off in the design space, given the different underlying hardware being controlled. The infrastructure properties coupled with the new requirements of situation-awareness applications cause state-of-the-art to lack some performance and functional requirements.

Situation-awareness applications have stringent requirements due to their inherent interaction with the physical world. For example, these requirements involve both the need for low latency access to computational resources and for geospatially colocated clients to be aggregated and processed together. Situation-awareness applications have upper bounds on the E2E latency of the processing of each measurement (see the previous augmented reality example). If the results come after an acceptable time window, they cannot be used to take action in the physical world. Similarly, many situation-awareness applications need to aggregate information from multiple clients in the same geographical locality (as the autonomous car example) , which requires the control plane to have global knowledge (*i.e.*, across all μDCs) to correctly select the application instance to handle a specific client (*e.g.*, sensor). These requirements highlight that scheduling tasks in appropriate locations and with the right partitioning of clients is critical for applications to maintain correct functionality.

The selection of geo-distributed resources and mapping application instances to clients

are key aspects of control planes for situation-awareness applications. Additionally, situation-awareness applications are highly dynamic (*i.e.*, workload spikes) and mobile, leading to the current application instance associated with a client becoming functionally unsuitable. These requirements highlight the need for a control plane that is agile and can quickly react to changes (*e.g.*, user mobility). The clients' context changes, like mobility, can affect both the E2E latency and the instance that should be processing the client. The control plane should, for example, update the mapping of clients to application instances to maintain the quality-of-service (QoS). Unfortunately, a traditional datacenter-oriented design cannot achieve these requirements as it does not collect the right metrics, does not understand geographical information, and is bound to communicate through slow wide-area network (WAN) networks.

These limitations raise the main question that guides this dissertation, which can be stated as: *How do we design an efficient control plane architecture for managing situation-awareness applications running on both geo-distributed micro and cloud datacenters?*

1.1 Problem Statement

Current processing control planes were not designed to handle hundreds of small geo-distributed datacenters, which we call μDC due to their relatively reduced size. State-of-the-art control planes have two assumptions on the computational infrastructure being used that degrades their performance under a geo-distributed μDC scenario: (i) a small number of big data centers and (ii) a low-latency interconnect between most resources. For geo-distributed resource and situation-awareness applications, these assumptions negatively impact the efficiency of the overall control plane architecture.

Centralized architectures are unsuitable for geo-distributed resources and situation awareness applications. The architectural challenge arises from the centralized control plane design on most current data processing frameworks, unsuitable for densely geo-distributed μDC [9]. A centralized design causes a potential high and non-deterministic latency between the control plane and the resources it manages and makes the control plane slow to respond to requests and react to changes. Situation-awareness requires agile responses to changes because applications are continuously evolving, given that the users are moving (e.g., augmented reality) and working with time-varying workloads, like rush-hour traffic captured on camera deployments. Propagating monitoring information to a centralized location and sending reconfiguration requests to each micro-datacenter incurs unnecessary latency that can be addressed with a better and more decentralized architec-

ture.

The control plane needs access to better definitions of situation-awareness SLOs. For example, as previously mentioned, situation-awareness applications need to have bounded low E2E latency processing and the capabilities to correctly group applications' clients together based on the geographical location for processing. Unfortunately, state-of-the-art control planes like Kubernetes [10] and KubeEdge [11] do not have mechanisms to expose E2E latency requirements that take communication latencies into account. Similarly, there is no notion of pairing groups of clients to instances based on geographical locations in load balancing frameworks like NGINX or processing frameworks like OpenWhisk [12] or Spark [13] that are used to extend the capabilities of control planes to support routing to multiple instances.

Scheduling requirements are different than in the Cloud. The scheduler for geo-distributed micro-datacenters requires optimizing for more parameters than in a cloud datacenter. First, selecting a micro-datacenter over another significantly affects the response latency of the applications, which means the location of both the µDC and the client needs to be considered during scheduling; the complexity is exacerbated due to the bigger number of micro-datacenters than regular cloud regions, as well as the heterogeneous network connectivity. The second parameter to consider is the limited size of the micro-datacenters; state-of-the-art control planes are not designed to manage limited resources at multiple micro-datacenters (*e.g.*, reservation and load-balancing) and do not have the right abstractions to migrate services to maintain the quality of service required by situation-awareness applications. Finally, monitoring and scheduling need to be done continuously, given the dynamic property of both workload and mobility of clients, but state-of-the-art control planes do not monitor the metrics required (*e.g.*, E2E latency and clients' mobility) by situation-awareness applications.

State management becomes a bottleneck for non-centralized architectures. Once we start considering a non-centralized design (partially or fully decentralized) for the control plane, the infrastructure state management (*i.e.*, resource allocation state) becomes a bottleneck for efficient execution. Once decisions are not centrally taken, data will be distributed across multiple locations and involve round-trips to coordinate between components; each extra message will impact the control plane's performance. Therefore, the state management mechanisms need to be designed to reduce the number of messages complexity in the system.

These technical complexities make current control planes unsuitable for efficiently

managing hundreds of micro-datacenters to process situation-awareness stream processing applications. To fully leverage all these new technologies, it is necessary to incorporate the geo-distribution notion and the micro-datacenter nuances into the control plane, as well as the specific requirements of situation-awareness applications.

1.2 Thesis Statement

We can provide an efficient control plane by combining decentralized and centralized components into a hybrid control plane architecture to provide both the low-latency benefit of decentralization and the global knowledge of centralized architectures. Furthermore, an efficient hybrid design can be accomplished by leveraging the infrastructure topology and application semantics to better map the functional components to the geo-distributed infrastructure. Finally, the hybrid control plane design will need two main building blocks. First, efficient state management and coordination mechanisms will be needed to combine the decentralized and centralized components. Second, a programming model capable of exposing the application nuances to the control plane to fully cater to the requirements of situation-awareness applications.

1.3 Contributions

This thesis makes the following contributions:

A programming model capable of expressing situation-awareness application and its requirements. First, we present a taxonomy of situation-awareness applications and use it to characterize the requirements of such applications. Then, we propose an extension to the dataflow graph (DFG) model to support the definition of application requirements. The programming model exposes intuitive interfaces to the application developer to facilitate latency- and location-sensitive application deployments. Additionally, the proposed programming model provides communication APIs for components and event handlers that facilitate the programming of the applications abstracting away both the geo-distributed nature of the infrastructure and the multiple instances of the application that will be deployed. Finally, the programming model permits decomposing applications into multiple components and independent logical partitions, letting the control plane better utilize the available resources, as it gives flexibility for the placement, partitioning, and replication of the application across the geo-distributed infrastructure. The programming model as the interface between the developer and the control plane defines the foundation on top, guiding

the definition of the control plane architecture.

A description of the requirements and components needed to support geo-distributed resources running situation-awareness applications. Based on a solid understanding of the application needs and the properties of the geo-distributed infrastructure, we describe the requirements that a control plane needs to fulfill. These requirements then guide the design of an architecture and the logical components that any such control plane needs to implement. Then, we perform an in-depth analysis of the role of each component and its interactions, as well as a description of the effects on the distribution of the components across the infrastructure. Defining the logical components of the architecture facilitates understanding of possible implementations of the components and their limitations.

An efficient control plane architecture and implementation of mechanisms to support the requirements. As a first step, we built and evaluated two control plane architectures for geo-distributed resources, centralized and decentralized, including the trade-offs in implementing the control plane's components. Our analysis focused on multi-tenant designs that can handle multiple applications collocated in the same µDC resources. Then, we progressively build the control plane components required for supporting all the needed requirements in a decentralized architecture. Accordingly, we propose decentralized mechanisms to discover µDC resources automatically and deploy application components onto the infrastructure commensurate with the application's latency requirements. Then, to continuously support the latency requirement and efficient use of resources, we present decentralized mechanisms for latency- and workload-driven resource allocation updates and migration of applications over space (*i.e.*, geographic) and time to deal with the dynamism in situation-awareness applications. Finally, we show the efficacy of the mechanisms with evaluations against situation-awareness mock applications.

Based on the learning and limitations of these two architectures, we build a hybrid control plane that combines the best of centralized and decentralized mechanisms into a hybrid control plane. The hybrid control plane combines autonomous decision-making at each µDC to minimize deployment latency for applications with tight-latency requirements, and centralized decision-making for scheduling applications that need efficient geospatial grouping. Furthermore, the hybrid architecture allows for better handling of high inter-component communication latency with an agile response to changes that could cause violation of quality-of-service requirements. It achieves this by reducing the impact of performing geo-distributed coordination with our proposed efficient optimistic concurrency control—the coordination is needed because geospatial grouping can cause the application

to span multiple μDCs. The concurrency control algorithm uses an enhanced 2PC protocol that leverages application semantics and infrastructure knowledge to reduce the common-case response time to one round-trip time (RTT) instead of the usual two RTT required by common coordination protocols for scheduling decisions that need global knowledge. Additionally, we implement a monitoring layer that ensures that each application's E2E latency and geospatial SLOs are met, triggering a client's migration (e.g., a mobile vehicle) to an appropriate application instance that aligns well with the client's latency requirements, as well as for changes in the geospatial group to which the client needs to be associated.

1.4 Roadmap

The remainder of this document is structured as follows, starting with chapter 2, which presents the core concepts required to understand this dissertation. Next, chapters 3 and 4 focus on laying out the foundations for building a control plane. First, we describe a taxonomy for situation-awareness applications and then use it to design a programming model tailored to those situation-awareness applications running on geo-distributed resources. Then, we focus on the requirements and challenges for a control plane in the context of this work and present the logical components required to implement it.

In the following chapters, the dissertation progressively builds the components of the control plane:

1. Chapter 5 describes a current state-of-the-art centralized architecture and describes quantitatively and qualitatively why it is unfit to manage situation-awareness applications in a geo-distributed setting.

2. To overcome some of the limitations of a centralized architecture, chapter 6 presents a fully decentralized architecture capable of providing E2E latency requirements and low latency decision-making.

3. Building on both the decentralized and centralized architecture, chapter 7 explains a hybrid architecture that allows overall better performance and provides all the SLOs required by situation-awareness applications, which was not possible with the other two architectures.

In the last part of this dissertation, chapter 8 presents the related work and describes its connection with this thesis. Chapter 9 broadly discusses the ideas and lessons learned in this dissertation. Finally, chapter 10 concludes this work and illustrates future directions of research.

CHAPTER 2
BACKGROUND

This chapter provides an overview of some terms and definitions required to understand this dissertation.

2.1 Geo-distributed Resources

Cloud is continuously evolving to provide better capabilities to the applications running on it. One of these trends is the creation of multiple datacenter (DC) regions, like Microsoft Azure [2] and Amazon AWS [14] regions. Multiple regions allow improving access latency, fault tolerance, and regulatory compliance. It allows having closer compute resources to the users and reduces the size of the network size to be traversed. Infrastructure users can deploy an application across multiple regions, reducing the likelihood of correlated failures.

The next step in this evolution is the deployment of computational resources colocated with 5G wireless infrastructure, such that the latency is cut considerably, with much higher bandwidth access to the resources. An example of this colocation is AT&T+Microsoft Edge Zones [15, 16] and startups like VaporIO[17], which are currently being deployed in multiple metropolitan areas.

We envision this trend to continue such that we obtain densely geo-distributed resources (i.e., resources locally available in populated areas close to users). Each of these sites would have a smaller footprint [7] than regular data centers. We dub them µDCs due to this reduced size. We expect them to be at least two racks of server-grade hardware from our discussion with the industry [8], but their size would vary depending on the expected demand in the area. For the remainder of this dissertation, when referring to geo-distributed resources, we refer to the combination of regular DCs and densely geo-distributed µDCs.

2.2 Situation-awareness applications

Situation awareness involves recognizing objects/entities in the environment and the relationship between them. A situation-awareness application derives actionable knowledge from sensors and client data. It involves three main steps: perceiving the environment, understanding its components and relationships, and sometimes also predicting the future.

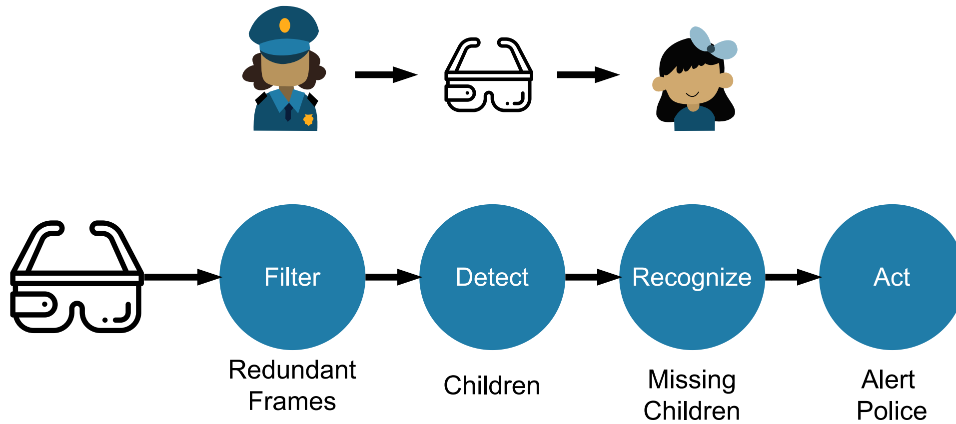

Figure 2.1: Situation-awareness application example and dataflow graph: police officer finding a missing child

Finally, the knowledge generated can be used to react and perform changes to the environment. Examples of sensors used as inputs are IoT sensors, cameras, and AR headsets. Due to its interaction with the environment, it requires low-latency processing to be useful. An example of a situation-awareness application is a police officer trying to find a missing child using an AR headset (*e.g.*, Google Glass), as shown in Figure 2.1. The Google Glass captures images of the surrounding of the police officer, and the application detects all the children in the frame and tries to recognize the missing children. If a child is found, it sends a visual alert back to the police officer to react appropriately.

2.3 Programming models for situation-awareness applications

Situation-awareness applications continuously process streams of data coming from sensors (*e.g.*, Google Glass). This type of processing matches the design of stream-processing frameworks [18, 19]. One of the primary interfaces for programming stream-processing frameworks is a DFG [20, 21]. A DFG is a graph where the nodes represent processing components, and the edges indicate data flowing between the different processing components, as shown in Figure 2.1. DFGs are a suitable programming model for this context due to their capacity of semantically splitting the computation and allowing each node to be placed independently. Additionally, DFGs are specially fit in our context because they can be augmented to contain application requirements at different stages/nodes in the graph (*e.g.*, latency requirements). An example of a DFG is in Figure 2.1, and the template ap-

plies to many domains. First, the application filters the incoming data. The following stage detects any event of interest. The third stage recognizes the objects found using the interesting events. Finally, the application creates connections and inferences between the detected objects and takes any required actions.

CHAPTER 3
PROGRAMMING MODEL

This chapter describes a general programming model for situation-awareness applications running on a geo-distributed infrastructure. First, it introduces a two-class taxonomy for situation-awareness applications and describes the requirements of such applications. Afterward, it formalizes a programming model that can express these applications and requirements and facilitate the work of developers implementing geo-distributed applications. Finally, it explains the interaction between the programming model and the control plane, including a discussion of the importance of the programming model in the efficient use of resources. This chapter has contents previously presented in OneEdge [22] and Foglets [23].

3.1 Situation-awareness applications

Situation awareness applications use a sensor fabric to convert information to intelligence. Sensed data comes from both mobile and static sensors, and the generated knowledge is used to act in the physical world (e.g., activating actuators) at computational perception speeds. Examples of situation awareness applications include emergency response, disaster recovery, and traffic congestion management. The application's objective is achieved by refining the input data and potentially aggregating information from multiple clients.

One of the critical enablers for situation-awareness applications is the recent ubiquity of connected hardware devices, often referred to as the IoT. The connected hardware, including robots and smartphones, is becoming more capable in every subsequent generation, getting us closer to the connected world imagined by the work of Satyanarayanan [24] and the ubiquitous computing vision imagined by Weiser [25]. Given the capacity of this new hardware, now we can derive more interesting insights from the sensors. However, to fully leverage these newer hardware capabilities, we need more complex applications that now will be geo-distributed, latency-sensitive, data-intensive, involve heavy-duty processing, run 24/7, and result in actuation over the physical world and possible re-targeting of sensors (*e.g.*, tilting a camera) at computational perception speeds.

3.1.1 Types of applications

We categorize situation-awareness applications into two classes: *standalone* and *coordinated*. This taxonomy makes understanding the requirements of these applications easier, in turn aiding in the creation of a control plane to meet their unique needs.

The most straightforward class of situation-awareness applications involves processing the information stream from one client and using the knowledge obtained to perform an action affecting the surroundings of that client. Usually, this sense-process-actuate loop would have stringent latency requirements such that the resultant action is still beneficial upon completion (e.g., moving a camera to point to an object of interest before that object leaves the area). This kind of application is also common when implementing control loops, such as those found in robot control (e.g., drone navigation).

The second most common class of situation-awareness applications involves aggregating information from multiple clients/sensors to enhance the overall view of the environment and improve the decision-making of each client. An exemplar member of this class is an application that enhances the view of autonomous cars within the same intersection. More specifically, when a set of autonomous cars reaches the intersection from multiple directions, the application will aggregate all the object detections performed by each car and return a complete global view to all the vehicles, allowing the detection of objects that could be occluded from the view of certain cars.

Since these two types of applications are representative of many situation-awareness applications, we classify situation-awareness applications into two categories: *standalone* and *coordinated*. *Standalone* is a generalization of the first single client exemplar application, and *coordinated* extends the multi-client scenario. Additionally, we can compose more complex applications with multiple building blocks of each of these types of applications.

Coordinated applications

Coordinated applications process information from multiple clients that are in close geographical proximity and aggregate their sensed data to either improve the accuracy of decisions or to coordinate across them. Examples of *coordinated* applications include collaborative assisted driving and geo-distributed multiplayer games (*e.g.*, Pokemon Go [26]).

To fully understand the *coordinated* category, we will describe the overall flow of the collaborative assisted driving application in Figure 3.1. In this application, each vehicle (*i.e.*, client) uses a lidar sensor and onboard processing to detect objects and generate a

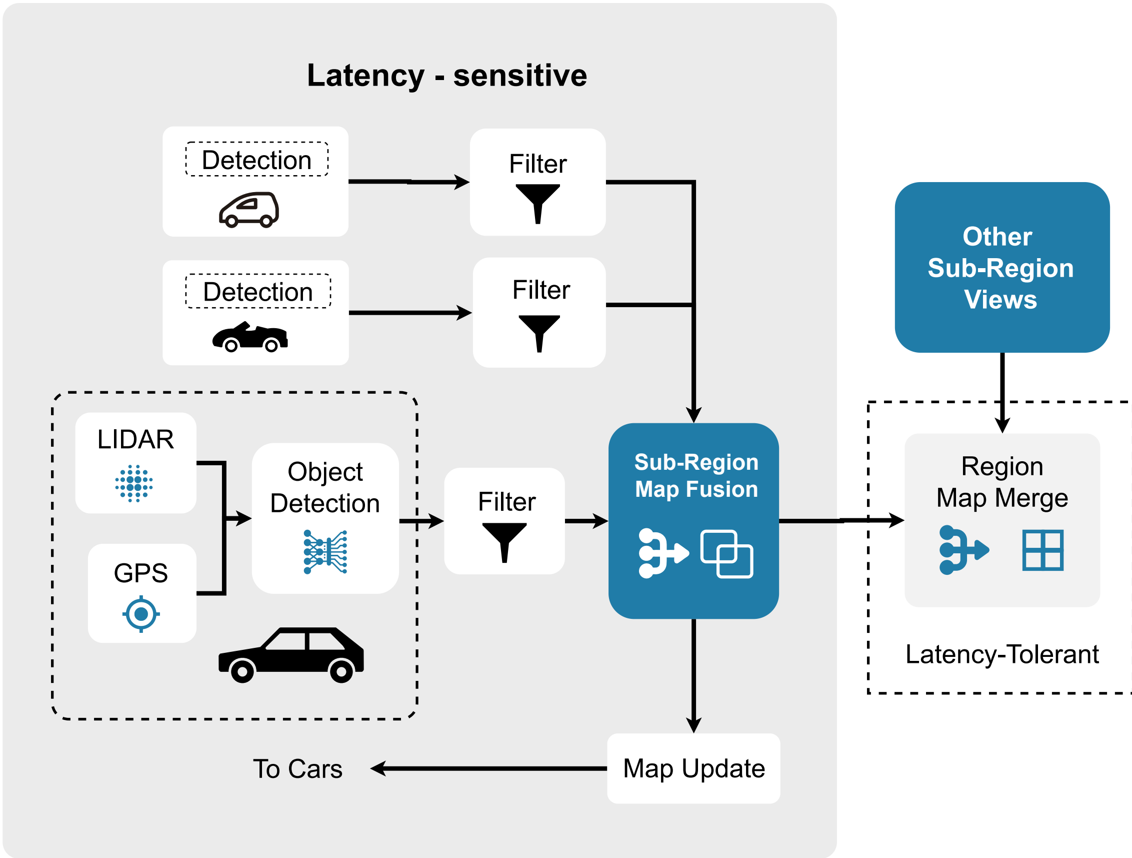

Figure 3.1: An exemplar of situation awareness applications – Connected Vehicles, a coordinated application. Cars in the same spatial locale have their individual views fused by the sub-regional view; the regional view fuses sub-regional views of adjacent spatial locales. The sub-regional view also sends feedback to the cars in a latency-sensitive manner.

list of all objects located in its immediate field of view. Then, the application aggregates individual views from multiple vehicles in close spatial proximity to one another to create a composite view (*sub-regional view*), which helps reveal objects missed by the individual views due to occlusions.

The fused list is used by two application subcomponents with different latency requirements. First, the fused object list is made available to the vehicles in the same spatial proximity so that each vehicle can make better decisions for lane control and collision avoidance; both of them are only useful with a bounded latency from each other. Second, fused object lists from disjoint regional areas can then be aggregated at the next pipeline stage to create a global view to improve vehicular safety and traffic pattern analyses.

This exemplar application shows how a situation-awareness application can have both latency-sensitive and latency-tolerant components. The latency-sensitive component requires co-location and proximity of computation (network latency wise) to the users to avoid network bottlenecks and provide the required latency requirements.

Another type of association required is concerning the locality of the clients (*i.e.*, cars). Each sub-regional manager is in charge of processing the information for a specific spatial region, and all vehicles in that region should send their local object detections to that same instance. Then, the mobility of the vehicles necessitates dynamic, constantly evolving associations of vehicles with spatial regions.

Standalone applications

In contrast to *coordinated* applications, *standalone* applications are limited to single-clients instances. Examples include augmented/virtual reality and single-drone control. In addition, latency requirements tend to be much tighter than coordinated applications as they can involve tight control loops instead of augmenting client data streams.

There are many reasons for offloading the computation from devices and sensors to the edge. The most common ones are energy, hardware capacity, and improving computational efficiency. For example, drones can be made smaller and with fewer hardware capabilities (or can reduce power consumption) by offloading the heavy detection and planning tasks to the edge micro-datacenters. Similarly, the data from small sensors (with cheap micro-controllers) can now be processed by much more accurate and complex machine learning models to obtain insights from the environment.

A good representative of *standalone* applications is the navigation control of a single autonomous drone [27, 28] in Figure 3.2. First, the drone streams its sensor data (*i.e.*,

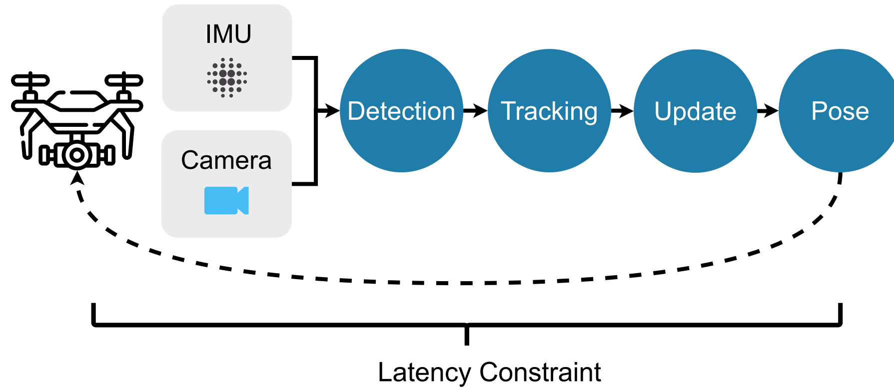

Figure 3.2: The standalone application process individual drones to calculate their pose based on the sensor inputs (camera and inertial measurement unit). The pose information is then feedback to the drone to continue its navigation.

camera and inertial measurement unit) to the application. Then, on receiving the sensor data, the application can detect the drone's location and indicate the following location to which it should move and the close-by potential obstacles. In this scenario, after sending the sensor data to the close-by application, it would be waiting for the message before taking the subsequent action; if the latency is too high, then the drone would not be able to make progress, and the control loop for navigation may become unstable.

3.2 Towards a programming model

The design of programming models (both languages and libraries) is both an art and a science. Indeed the history of programming languages/libraries (sequential, parallel, and distributed) is as old as computer science itself. Examples of successful programming languages include C/C++/Java/Python, and programming libraries include pthreads and MPI. The "art" attributes for success are simplicity and ease of use for the developer (i.e., the domain expert). Needless to say, some elements of "luck" and "timeliness" play a part in the art attribute as well, which would explain why some elegant languages (e.g., Modula/Algol/Pascal) did not survive the test of time. The "science" attribute is the efficiency of execution of the model commensurate with the application domain's needs.

The development of a programming model for situation-awareness application undertaken as part of this dissertation research builds on the rich history of prior art. The taxonomy of situation-awareness applications presented in section 3.1 can be used as a guide

for designing such a programming model. We hasten to add that the intent in this dissertation is not to strive for syntactic or semantic elegance of the programming model as would be the focus of a programming language dissertation. It is merely to serve as a good starting point for translating the requirements of situation-awareness applications to an intuitive and actionable application programming interface (API) for the domain expert. Such an API would serve as a prescription for the design of the control plane for mapping situation-awareness applications to the edge-cloud continuum of computational resources.

Drawing from the application needs in section 3.1, we identify three main attributes of the programming model:

1. It should be tailored for continuous data streams, given that sensors are constantly generating data.

2. It should allow the definition of E2E latency requirements, as standalone applications require bounded latency of the control loop that manages the clients' behavior.

3. It should support the description of how clients are functionally grouped in coordinated applications. The grouping of clients is application-specific as it depends on the speed of mobile clients and the relative importance of different geographical areas. For example, in the connected cars application, a busy intersection is more meaningful than a segment of a long lone street.

The programming model should impose minimal burden on the developer (i.e., ease of use) while providing enough expressiveness to represent a diverse set of situation-awareness applications.

At the same time, the disaggregated nature of the infrastructure suggests that the model should enable the control plane to make scheduling decisions respecting the scarcity of resources in any given edge site. To facilitate such decision-making, the programming model should provide sufficient information via SLOs.

A streaming programming model that captures the continuous processing endemic in situation-awareness applications is the starting point for the exploration of the design space. The next observation regarding the SLOs of such applications is that not all the application components have stringent latency requirements and, therefore, could be provisioned in the cloud to reduce the resource pressures on the micro-data centers.

In order to allow the partitioning of applications, the simplest model would be a pipeline of components, where different components could be assigned different requirements, and

the control plane would be able to deploy them judiciously on the right resources. However, it is hard to enforce separation of concerns in a multi-stage pipeline, as our target applications possess multiple sensor streams, and they would need to be processed by the same pipeline stage. For example, the drone application will send information about both the camera and the inertial measurement unit. This limitation creates harder-to-understand code as logic for different sensor types must be handled in the same component. A better approach is to use a general dataflow graph, where each node in the graph can be assigned a unique task and only need to output a well-defined output stream to be consumed by downstream nodes. Such a design allows for separation of concerns [29] and facilitates differentiated requirements to use the available resources judiciously.

The definition of E2E latency bounds for situation-awareness applications is different than for datacenter applications. Situation-awareness applications require that E2E latency be bounded, including the communication back and forth from the μDCs/DCs. An initial approach would be to let the developer define the execution latency for each application component, but such an approach would be too fine-grained and, most importantly, may not match the application's needs. For example, a typical situation-awareness application needs to bound the time between *generating* a data item from a sensor to its *consumption* by a given app component, rather than specifying the per-component execution latency. Application developers' burden can be significantly reduced by letting the control plane be responsible for determining the appropriate execution/processing and communication latencies such that the application's needs can be satisfied. Therefore, a better abstraction of this need would be the *staleness* of the input data at a component, calculated from when the initial sensed data was generated. The abstraction should also allow specifying data staleness constraints for the information being fed back to the client from the application components. The staleness then describes how long it took to generate and communicate an event at an *input* to a given stage of the dataflow graph. This metric matches the expectation from the developer and allows for defining it only for those stages that are sensitive to E2E data staleness.

Coordinated applications require a way to specify the functional grouping of clients to aggregate their sensed data. A starting point to satisfying this need is to let the control plane be fully in charge of performing the grouping by spatial proximity. For example, the control plane could perform clustering of the clients and assign them to different instances. However, this design fails to provide the application-specific semantics of different geographical areas, as in the example before, with the difference between a busy intersection

and a long lone street. The developer should be able to specify the regions of interest in a programmatic manner. One possibility is to explicitly split the physical world map into non-overlapping polygons defined as a configuration file. However, such an approach requires an explicit description of each region, and it is hard for the developer to make changes later. To improve the ergonomics of the interface, a better option is to let the control plane use a developer-provided mapping function that converts a GPS location to a unique identifier. This function facilitates evolution from a developer's perspective and does not require an exhaustive description of all regions in the world while still giving the developer the flexibility to express the semantics better.

In the following sections, we formalize these ideas into a programming model based on a dataflow graph construction and support the application requirements of (a) E2E latency and (b) spatial grouping, via *data staleness* and a *programmatic location mapping*, respectively.

3.3 Programming model

Situation-awareness applications generally process data streams coming from sensors and users. These streams are processed to generate actionable insights for the clients. A DFG can naturally model this flow of information. Each node in the DFG is an application component. Similarly, the directed edges represent the flow of information from one component to the next. A special case of the DFG is a pipeline of components, which also is common for situation-awareness applications. For example, the connected car application presented previously can be modeled as a pipeline of application components, as shown in Figure 3.1.

In the connected car application (Figure 3.1), the client (*i.e.*, car) feeds input data into the DFG. Each component (node in the DFG) processes data generated by the upstream component (or the client) and generates output data to be consumed by the downstream one; in Figure 3.1, the first component receives input from the cars, filters out unimportant detections, and streams down to the next component (sub-regional view). Finally, any component can send actionable data back to the client directly. For example, in Figure 3.1, messages can be sent back to the client from both the sub-regional and the regional view.

More formally, each DFG's node is a logically independent actor (similar to Orleans' agents [30]). A node reacts to events from connected nodes (section 3.5). For example, the first node (*i.e.*, filter) will react to either an incoming message from the client or a new client's appearance.

Additionally, the DFG partitions the application logically into different functional com-

ponents. This partitioning helps separate the service-level objectives required by each component, as not all components have stringent requirements; we discuss these requirements in section 3.4.

3.4 Application requirements

From the application taxonomy, we can notice three primary requirements for situation-awareness applications: E2E latency, bandwidth, and spatial (geographical) affinity. First, the E2Es latency of the application execution needs to be bounded to be helpful to the client (*e.g.*, the latency to detect objects for collision avoidance in the connected car application). Second, both types of applications have stringent bandwidth requirements given that they are continuously streaming potentially raw sensor data (*e.g.*, video frames) and, for example, could potentially be restricted by the available wireless spectrum. Finally, *coordinated* applications require that clients in the same geographical area be aggregated together in a timely manner. In this section, we analyze in more detail these application requirements and how they can be formally expressed as part of the programming model to define the QoS required by the application.

3.4.1 End-to-end latency SLO

End-to-end latency SLOs represent the maximum allowable latency for the full execution of the application. Specifically, situation-awareness applications have a time window for which a result is useful after the original data is generated. If the response comes after the given window, no meaningful action can be taken. For example, in the connected car application, if the bounding boxes are returned after the car was at the location of the bounding boxes, they are no longer useful for improving decisions. On the other hand, if the drone does not timely receive the navigation control decisions, the control loop can become unstable, and the drone may not reach its final destination—or worse, crash [31]. It is for this reason that E2E latency SLOs are better defined in terms of tolerable staleness of data, *i.e.*, how stale is the input data that was used to generate this specific output.

In section 3.3, we proposed modeling situation-awareness applications as DFGs. Then, to leverage the topology structure of the DFG, the latency requirements should be specified per application component. An example of how E2E latency SLOs can be defined is shown in Figure 3.3. In this example, the *Sub-Regional View* stage has a tighter latency requirement than the *Regional View* stage, given that one is used to take real-time decisions while the other is used for longer-term optimizations. This example also illustrates how latency

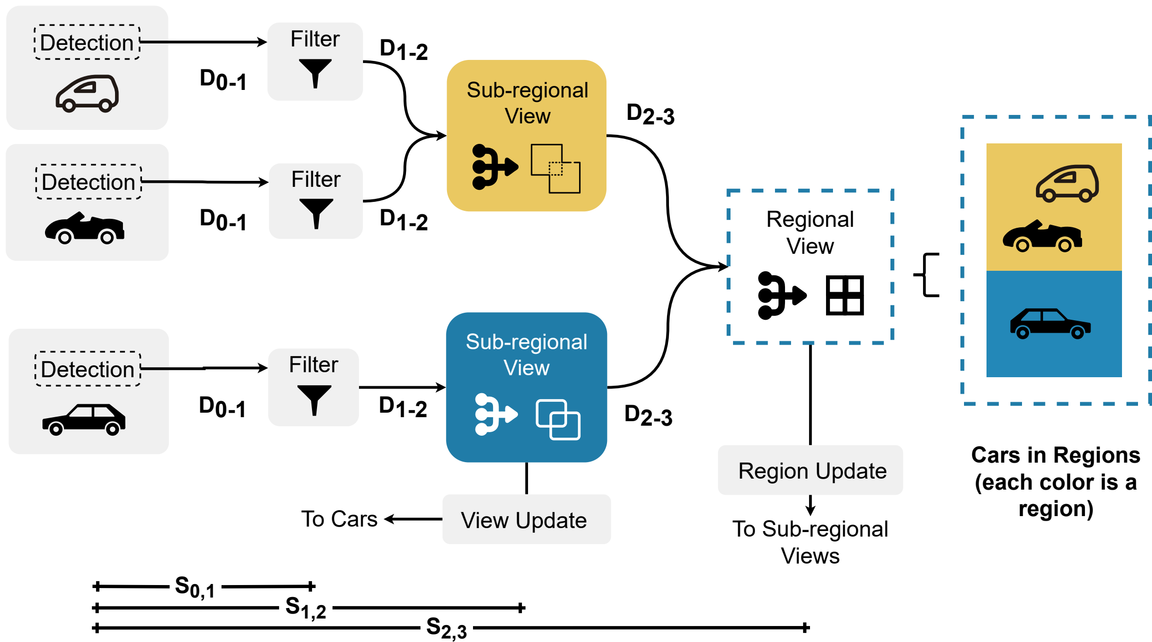

Figure 3.3: The connected car application is annotated with three different types of service-level objectives. The sub-regional views have two colors associated with different areas of interest, where two cars are in the yellow region and one car is in the blue region; corresponding instances manage each of the spatial regions. The $S_{i,j}$ presents the end-to-end latency requirements of the corresponding j stage coming from node i. The $D_{i,j}$ presents the bandwidth requirements for the link going from node i to node j.

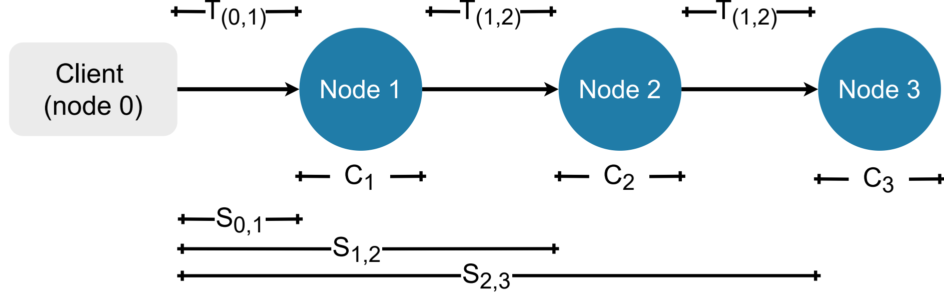

Figure 3.4: A generic pipeline that explains the different components of the tolerable latency staleness. $S_{(i-1,i)}$ is the acceptable latency starting at the output of the client to the input of the node i. It is composed of both computational latency C_j and transmission latency $T_{j-1,j}$.

requirement depends on the application domain and needs to be defined by the application developer.

More formally, Figure 3.4 shows the tolerable latency staleness for the application component i using the notation $S_{j,i}$. $S_{j,i}$ denotes the worst-case acceptable composite latency at the *input* of application component i coming from the output of node j. $S_{j,i}$ accounts for all the upstream processing and communication times from generating a message at the client until stage i's input going through node j. Concretely, the E2E latency SLO of an application modeled as a pipeline is equal to:

$$S_{(i-1,i)} = \sum_{n=1}^{i-1} C_n + \sum_{n=0}^{i-1} T_{(n,n+1)}, \tag{3.1}$$

where i is the stage id of the node for which the latency staleness is being calculated (with $i >= 1$), with id 0 being the client. Additionally, C_i is the computational latency of stage i, and $T_{(i,j)}$ is the communication latency between stages i and j. Equation (3.1) assumes no queuing; if required, an extra summation is needed to account for it. In the general DFG case, the latency is calculated for all paths from the client to that specific application component id i, which goes through the link between j and i, and the maximum across all the latency calculations to check if it is providing the required bound.

Two properties may seem counter-intuitive from this metric definition. The first one is that it is calculated up to the input of an application component and not the component's output. This decision was taken to make the last communication latency part of the stale-

ness calculation. So, for example, the messaging back to the client from the *sub-regional view* should be incorporated as part of the staleness of the calculation. The second one is why the staleness $S_{(j,i)}$ is tied to a specific edge in the DFG and not just node i. The reasoning is that multiple feedback loops could go back to the client, and each one may have different latency requirements. For example, in the connected car application, the *sub-regional view* requires low latency, but if we add a higher level *geo-routing node* to the application, that could have looser latency requirements. A DFG does not allow multiple edges between nodes, so this is a unique way to specify the E2E latency.

3.4.2 Bandwidth

The bandwidth requirement can vary drastically at the input of the different application components of an application. For example, in Figure 3.3, the *Filter* component receives the raw data stream from the onboard sensors and needs a high-bandwidth connection, while the *Regional View* stage, which aggregates summaries from sub-regions, has a considerably smaller bandwidth requirement.

Similarly to the E2E latency requirement, we can define the bandwidth requirement per application component as the output *rate of data* production. D_{i-j} indicates the production rate of data items communicated between the application components *i* and *j*. For example, the data rate between the *filter* and *sub-regional view* application components D_{1-2} is related to the objects detected per unit of time (bytes/second).

Based on their domain expertise, the developer can define the expected bandwidth requirement between a pair of applications components in the DFG. The bandwidth requirement is the minimum required for correct functionality. For example, if the developer knows that the application will handle 4K video continuously 24/7, then they can specify the requirement of the edge between the camera input stream and the first level of processing to be the minimum bandwidth that the camera is going to use (*e.g.*, 10 Mbps [4]). This requirement forces the control plane to choose a resource allocation that can provide that bandwidth, reducing the likelihood of other types of violations.

3.4.3 Spatial Affinity

Coordinated applications also require defining *spatial affinity* across clients as an objective of interest. We defined *spatial affinity* in OneEdge as the *application's intent to share state among a subset of clients based on geographical proximity, i.e., area of interest (AoI)* [22]. Different applications will have a diverse way of establishing the AoI for their clients. For

example, in the connected vehicle application, the developer will probably separate each busy intersection as a different AoI. On the other hand, a geo-distributed multiplayer game would split the world concerning geographical features, like a shop or a skateboard park.

In order to specify the spatial affinity requirement of an application, the developer defines a function that maps a GPS location to an AoI, where each distinct AoI would have a distinct identifier (integer). This function can then be used to map a current client's location to the AoI to which it is supposed to be associated at a given moment in time. More formally, the developer defines a mapping function M, such that:

$$M : X \times Y \rightarrow \mathbb{Z} \tag{3.2}$$

$$X = \{x \in \mathbb{R} \mid \frac{-\pi}{2} < x < \frac{\pi}{2}\} \tag{3.3}$$

$$Y = \{y \in \mathbb{R} \mid -\pi < x < \pi\} \tag{3.4}$$

where X is the set that contains all possible latitude values, and Y is the set of all possible longitude values. In other words, M is a two-variable function that maps a given (latitude, longitude) pair to an integer that represents an AoI. Spatial affinity does not disallow co-locating multiple AoIs in the same instance (*e.g.*, sub-regional manager) but enforces that certain clients are collocated in the same application instance. The control plane then needs to guarantee the uniqueness of AoI deployments, as there should not be more than one instance of the application for a specific AoI identifier.

In contrast to the well-known bandwidth and latency requirement metrics, we had to define a new type of metric in Equation (3.5), which would guide the control plane to optimize deployments for the locality of clients. There may be periods when spatial affinity may be temporarily violated in real application deployments. For example, a car disconnected momentarily from the network and reappeared in a different AoI, but the client is still connected to the previous AoI. To quantitatively compare two potential mappings at a given moment in time, we define the *spatial alignment* [22]. The *spatial alignment* for an AoI can be defined as follows:

$$spatial\ alignment = \frac{\#\ of\ clients\ in\ AoI\ sharing\ the\ app\ pipeline}{\#\ of\ clients\ in\ AoI}. \tag{3.5}$$

A perfect spatial alignment would obtain a numerical value of 1, and all the clients with the same spatial affinity would be connected together. The metric value would decrease for each client assigned to a different AoI than the one prescribed by the function M.

Spatial misalignment can heavily impact the behavior of applications. Coordinated applications intend to improve the accuracy of the perception and understanding of the world for each client connected to a given AoI. For example, in the coordinated application shown in Figure 3.1, the views of multiple cars are fused such that occlusions and any limitation in each car's view can be improved to have a complete 360 view of its surroundings. When a car gets assigned to a wrong AoI, it is receiving updates for the information that is not relevant to it, but more importantly, its worldview is not being enhanced, which can cause unnecessary difficulties. For example, in the connected car application, these difficulties can range from reducing traffic flow efficiency to avoidable collateral damage due to occlusion. Another example is a collaborative augmented reality application, where the wrong AoI can break the illusion of the application, resulting in two clients looking at the same objects and receiving different updates. In general, wrong AoI matching degrades the quality of the application with potentially unnecessary harmful effects.

3.5 API and runtime handlers

For the application developer to implement the programming model, we propose using two main constructs: a low-level API and a set of event handlers. The low-level API exposes both the communication and persistent data management to the developer. For example, an application component (*e.g.*, filter in Figure 3.1) would call the "send" function with the corresponding serialized object to communicate to the downstream application component (*e.g.*, sub-region view in Figure 3.1). This communication event would trigger an event handler on the downstream application component, a handler also implemented by the developer. The control plane runtime invokes the event handlers upon certain events.

The application code consists of a set of event handlers that the application must implement and a set of functions that applications can call (API). The use of these two constructs facilitates the developer's work (section 3.7) and improves the efficiency of the control plane (section 3.8). Additionally to the programmable low-level API and handlers, the user also needs to declare the topology of the DFG of the application and the application requirements (section 3.4) as configuration files. The use of handlers and APIs is a natural representation of a DFG, as they represent the flow of information similar to how DFGs are logically constructed, where nodes send data to downstream nodes, and the downstream

nodes process those messages. Additionally, it adds a layer of abstraction (*e.g.*, when to process the events and how to send the data) that the control plane can leverage.

In the remaining parts of this section, we first present the logical partition of applications based on the application types (section 3.1.1), and then we present the core APIs used by the developers to implement situation-awareness applications. Finally, we split the discussion into two logical components: communication (section 3.5.2) and data access (section 3.5.3).

3.5.1 Logical partition of applications

Given the geo-distributed aspect of situation-awareness applications and their clients, the programming model should provide hints to the control plane to partition the applications to be more efficiently executed in the available geo-distributed infrastructure. Both the application types and the topology of the DFG provide a good guide to partition the applications' clients logically.

In *standalone* applications, each client is independent of other clients. This property allows for partitioning each client as an independent logical instance. On the other hand, the granularity of partition in the coordinated applications is the AoI. Coordinated applications group clients together based on the AoI defined by the developer and provide a clear division across multiple clients. Additionally, since each application component in the DFG is an independent agent, it also allows the splitting of each component in the DFG to be run independently. The control plane will need to deploy multiple independent instances of an application DFG to support the application requirements (section 3.4); the logical partitions will guide the deployment of the multiple instances.

The two axes for partitioning (application type and topology) affect both the control plane's actions and the abstraction used to define the state management; we discuss the state management (section 3.5.3) and how the mapping from partitions to actual physical executions (section 3.8) in subsequent sections. The logical partitions also help with the migration of application components between µDCs, given that it gives a clear definition of which clients need to be moved together, which is discussed in later chapters of this dissertation (chapter 4).

3.5.2 Inter-component communication

Each of the logical application components in a DFG can communicate with each other using a hierarchical communication API, namely, *send up* and *send down*, as well as a

Table 3.1: Programming model API: communication primitives.

Interface	Description
void send_up (message m, edgeId o)	Sends a message asynchronously from a node to the downstream node connected through edge *o*.
void send_down (message m, edgeId i, **optional** nodeId n)	Sends a message asynchronously to all upstream nodes connected through edge *i*. Optionally it can choose to only contact one of the upstream nodes *n*.
void send_to (message m, nodeId destination)	Sends a message to a specific destination node.
void send_to_partion_clients (message m, partitionId id)	Sends a message to all the clients in a logical partition.

point-to-point communication API. All the communication APIs and associated triggered events on the receiving end are shown in Table 3.1.

The hierarchical communication is used to directly represent the DFG of the application. For example, *sending down()* is invoked on an application node when a message arrives from an upstream node. Similarly, a node can reply to the upstream node by using *send up()*. Given that a DFG can contain multiple incoming edges, the hierarchical send also needs to specify the name of the logical edge in the DFG. However, given the already known topology of the DFG, it can avoid explicitly contacting a specific application component instance.

On the other hand, the point-to-point API allows any application component node to potentially connect to a different instance of any application component node. The main complexity of point-to-point communication is that identifying specific nodes for point-to-point communication should be disseminated through hierarchical communication. The number of instances for each logical application component is defined at runtime, and it is not possible to predefine the name of a given instance in a given location, as the control plane will handle this.

The main reasoning behind the explicit division between the hierarchical primitives and the point-to-point primitives is to expose the differential cost between the two primitives. We provide the hierarchical communication API to encourage application developers to perform more efficient in-network processing. The control plane will consider the application requirements, resource infrastructure, topology, and the application DFG to deploy it. The control plane will optimize the hierarchical communication for frequent inter-node communication, while the point-to-point communication will not be optimized. However, the point-to-point API is provided to the developer to have a consistent and simple out-of-band communication, which also leverages the runtime's asynchronous event handlers.

Table 3.2: Programming model API: manipulating the local object store.

Interface	Description
set<object> get(key k, partitionId pId, time t)	Get the application data that matches a type, partition id, and time range.
void put_object(object o, key k, partitionId pId, time t)	Put application data associated with a key, partition id, and time.

3.5.3 Data Management

We propose a best-effort data storage API to facilitate the implementation of situation-awareness applications with soft-state. An application component stores its application-specific data in a local object store called the spatio–temporal object store. More specifically, it provides a key-value interface, where the application component stores its key-value pairs tagged by key, time, and partition identifier, using *put object()* and *get object()*, as shown in Table 3.2. For example, a traffic monitoring application may store detected license plate numbers, tagged by the detection time and *LicensePlateNumber* key.

In general, the spatio-temporal object store allows the application to store the relevant spatio-temporal information such that it is available to the application even when the control plane takes actions (*e.g.*, a logical partition migration). An important aspect is that the object store guarantees are best-effort —and not guaranteed— as the application state can be recomputed from newer incoming data (*i.e.*, the state is soft). For example, when processing static camera streams, the application will normally generate a model of the background. If the state of that model is lost, it can be easily regenerated (with a time penalty) from new video frames. The soft-state is not functionally required, but its existence allows to avoid unnecessary recomputations when the data is still available in the geo-distributed runtime. If the data is not found in the object store, the application component will recompute it.

The partition identifier is associated with the logical partitions in section 3.5.1, which is passed as part of the callback for event handlings for each application component (Table 3.3). As a reminder, for standalone applications, the logical partition is associated with a specific client, and for coordinated applications, the logical partitions would be associated with a specific AoI. The partition identifier allows the control plane to manage the data concerning the logical partitions, with more details of implementation in section 3.8.

Table 3.3: Programming model handlers: invoked by the the runtime on message arrival.

Handler	Description
void on_send_up (msg m, partionId pId)	Called when a new message arrives from a upstream node.
void on_send_down (msg m)	Called when a new message arrives from a downstream node.
void on_receive_from (msg m, nodeId source)	Called when a new message arrives from a peer node.
void on_migration_start (partionId pId)	Called before a migration process starts. Application code running at the original instance should perform any required final operations before returning.
void on_migration_end (partitionId pId)	Called after a migration process ends. Application code running at a new instance can recover from the spatio-temporal store.

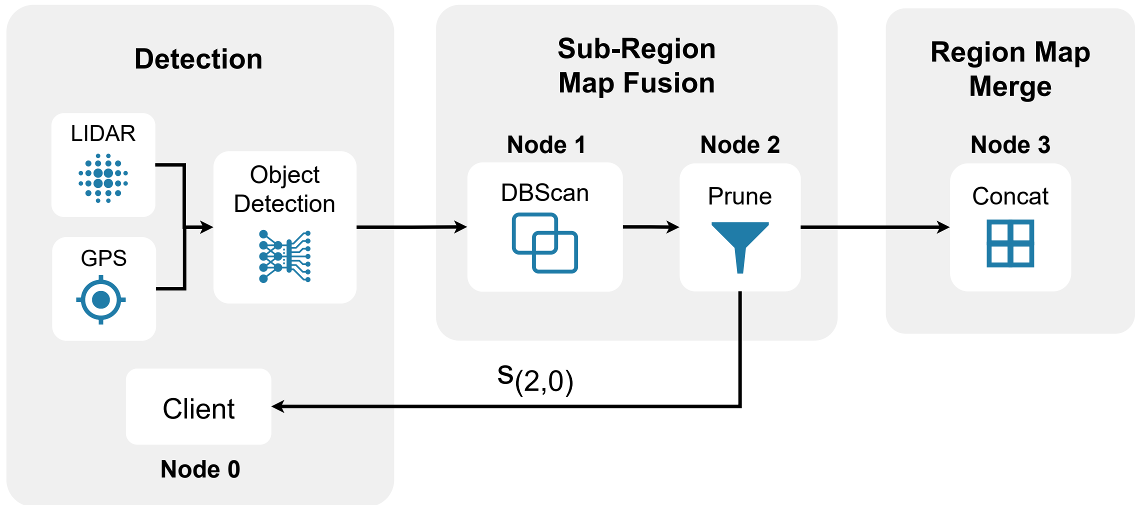

Figure 3.5: Connected cars dataflow graph. The dataflow graph has three stages: dbscan, prune, and concatenation. The first two stages are part of the *sub-region map fusion*, and the last one is part of the region map component. The last sub-region map fusion component also sends data back to the client. This application has two service-level objectives: a latency staleness bound $S_{(2,0)}$ of 100 ms and spatial affinity constraints for all three nodes, with different areas-of-interest for the third node than the first two nodes.

3.6 Example of an application implementation

This section presents the implementation of the coordinated car application in Figure 3.5. This figure decomposes the connected car application from previous sections into a DFG. The DFG has three main stages: *DBScan*, *prune*, and *concat*. The first stage (*i.e.*, object detection) is not counted as it is processed internally by the autonomous car. *DBScan* [32] is a well-known algorithm for clustering objects, in this application is used to find which objects overlap in the view of multiple cars. The pruning node then aggregates these groups of overlapping objects to obtain the final list of objects. *Pruning* defines which objects are independent, and for equivalent objects, it uses information from multiple observations to define the most likely orientation. Finally, *concat* takes the local sub-regional views and concatenates them to form a full region map.

3.6.1 Configuration file

The first step that the developer takes is to create the JSON configuration file. The configuration file comprises three parts: container configuration, the topology of the DFG, and application requirements. An example of this JSON file can be found in Figure A.1.

Each node in the DFG is labeled with a unique name (within the application) to describe the topology. These labels are reused in the code implementing the components. Using these names, the developer describes the edges as a list of tuples, and each tuple is composed of three elements: source node, destination node, and label. The edge label is used in case a node has multiple output edges. For example, in the connected car DFG, there would be two internal edges: (“dbscan”, “prune”, “objects”) and (“prune”, “concat”, “objects”), the label of the edge only has to be unique for each given source node. Additionally, to simplify the naming convention, we split the edges into two groups: one for communication between application components and another for communication between components to the clients. The final topology configuration is the node’s name that receives the messages from the client. In the aforementioned example, it would be the *DBScan* node.

The container configuration is simple, as it only needs to include the library’s location within the container image that hosts the application components. Then, the location is used by the runtime library process to load the library for executing the application component.

Finally, the application requirements include both latency and spatial affinity descriptions. The latency is defined as a pair formed by an edge tuple and the latency requirement in milliseconds, and we use the special keyword “client” to represent that destination node

for latency in edges facing toward clients. For example, in the connected car application, this requirement would be added as ((“prune”, “client”,“”), 100), indicating that there is a 100 ms latency requirement for the data staleness at the client’s input coming from the *prune* node. Client input edges do not have a tag name, as they use a different communication API. The spatial affinity description is a mapping between a node name and an identifier that the code will use to call the appropriate affinity function. For the exemplar application, both *DBScan* and *prune* nodes would be mapped to the “sub-region” function, while the *concat* node is mapped to the “region” function. This setup will become clearer when we explain the implementation of the code.

3.6.2 Code representation

Three different entities form the code that implements an application: the bootstrap function, the application components, and the spatial-affinity function. The bootstrap function initializes the required object to execute a specific application component. The application components are each an object that encapsulates the functionality of each node in the DFG (*e.g.*, the *DBScan* node). Finally, the spatial-affinity function takes as inputs the client’s location and the mapping identifier (*i.e.*, the one defined in the configuration file) and returns the AoI’s identifier.

The bootstrap function receives as input the name of the application component (from the configuration file). The function then registers the corresponding object instance that will handle all the requests for that application component. The bootstrap function can also initialize any other global variable or configuration of the object instance from documents available inside the container. For example, The bootstrap function for the connected car application will have two potential name inputs: *DBScan*, *prune*, and *concat*. Then, each of these inputs would create a different type of object. The pseudo-code for the bootstrapping process for the coordinated car application is available in the appendices (Figure A.2).

Each application component is implemented by extending a base “ApplicationComponent” class, which is available as part of the library that includes the communication and storage API. The developer implements all of the corresponding handlers (Table 3.3), using the programming model API (Table 3.1) and any other domain-specific libraries required by the node. For example, to implement the *Prune* node, the developer will need to implement the “on_send_up” to handle the messages coming from the *DBScan* node. Then, the handler will deserialize the incoming message, perform the application-specific logic to refine the list of objects, serialize the filtered list, and send it to both the client and

the region *concat* components. Finally, the handler uses the "send_to_partition_id_clients" function to send it to the clients and the "send_up" function to send it to the *concat* node. The pseudo-code for this application component is presented in Figure A.3.

All the other components of the application will be implemented similarly. The components usually have the following steps: deserialization, processing, saving state, serialization, and communication. However, some application components may have more complex implementations. For example, the *DBScan* node has two additional operations beyond the basics: state and migration.

The *DBScan* node needs to keep track of the cars being merged. Not all the cars send their results simultaneously to the *DBScan* node, so the node needs to perform temporal alignment on the input streams, and it should wait for all the cars' detections before aggregating them. To perform the alignment, it needs to know what cars it has seen in that AoI in the last few data cycles to wait for them. The car membership is stored in the spatio-temporal object store after each iteration. This data is important when a *DBScan* instance is migrated.

On the migration of an AoI to a new instance, the car membership data will be fetched such that when the cars start communicating with the new instance, the node knows for which of them to wait. The time input to the spatio-temporal request will be the maximum time the component can wait for a car before the latency bound is reached. The use of the spatio-temporal store and the implementation of the migration handler is shown in Figures A.4 to A.7.

The last component that the developer needs to implement is the spatial affinity function (with pseudo-code in Figure A.8). The function will receive the client's location and either "sub-region" or "region" as input for the connected cars application. A region will be composed of a group of neighbor sub-regions (*i.e.*, a region fully subsumes a group of subregions). The developer then selects the right way to partition the areas of interest into sub-regions and regions. For example, an intersection may be a sub-region, and downtown is a region.

This section exemplified how the configuration, handlers, and APIs help developers implement the applications and specify the requirements. In the next section, we discuss the effects of the programming model on the implementation of applications in more detail.

3.7 Effect on application implementation

A DFG allows separating the core application code from the mechanisms provided by the software infrastructure. Additionally, a DFG simplifies large–scale application development since a developer does not need to write different programs for heterogeneous devices with different connectivity. For example, the runtime will run the same application code on various devices, including clients and the computing nodes in the edge and the cloud.

The main drawback is that because each application component in the DFG is independent, sometimes additional information will need to be sent between them. However, the DFG provides greater flexibility to the control plane to efficiently place the components and handle the automatic scaling and replication of running instances, while still providing the application requirements.

The developer needs to provide three inputs to the control plane to submit an application for future use:

1. The representation of the DFG presented in the previous section.
2. The handler's code for each of the application components in the DFG compiled into a container image using the programming model library.
3. The application requirements for the corresponding application components. For each relevant component, the developer will provide any or all of the requirements presented in section 3.4. We designed the requirements to be easily defined per application component, as explained previously.

After submitting these three inputs to the control plane, the runtime will return an associated unique identifier called an app-key that will be used to identify this version of the application submitted. When a client connects to the infrastructure (either first time or after a disconnection), the client will provide the app-key, and the control plane will connect it to a corresponding instance of the application (and deploy it if required).

The combination of the API and the event handlers abstracts away all the complexities of geo-distributed systems from the developer. For example, the developer does not need to worry about where (*i.e.*, which of all the nodes in the infrastructure) the code is running; the control plane handles both the data and the communication between components. Given the tight latency and spatial requirements of situation awareness applications, the application components may be migrated to a closer computational node. After a migration, from the application's perspective, there is no change, the data is accessible (through the API), and

the corresponding blocks of code will be called via the event handlers. Similarly, running connected application components on different geo-distributed nodes is invisible to the application code, given that the message is also sent through the API, and the corresponding event handler is called in the other locality.

3.8 Effect of programming model on the control plane

Geo-distributed resources form a continuum fabric composed of the clients, the micro-datacenters, and the cloud datacenters. Therefore, there is a need for the right primitives to allow the control plane to place the application components (generated from the streaming programing model), move data among the components, and migrate computation and state commensurate with the sensor's mobility pattern (e.g., autonomous cars) along this continuum.

Abstracting the application into a DFG allows the control plane to perform certain optimizations that can improve the usage of resources. For example, the discovery and selection of computation for clients are hidden by a thin layer running in the clients. As a result, the control plane would automatically select the appropriate geo-distributed computing node to host an application component for processing the incoming stream data. The abstraction also allows per-client SLO-driven decisions. Presenting the SLOs in terms of data staleness and rates give flexibility to the control plane in placement decisions, taking into account the µDCs' resource capacities to accommodate each stage's CPU and memory needs and the bandwidth required for inter-stage communication. This use of objective-based resource allocation instead of explicit capacity selection also allows merging DFGs from different clients, discussed in more detail in the following subsections.

The DFG programming model also allows an additional level of indirection for the control plane to define when to perform the resource allocation. The first trade-off that the programming model allows is late-bind vs. early-bind. As in Sparrow [33], late-binding allows delaying the assignment of a specific resource for a client until they are needed. For example, *coordinated* applications would benefit from early-bind, such that the coordinated deployment of a specific application component instance for an AoI is done once, and it is accessible by other clients that arrive in the same area. On the other hand, the control plane has more control over when to allocate the resources in *standalone* applications. For *standalone* applications, there is another level of flexibility in the decision; the control plane can make allocation decisions per message (similar to what current serverless engines do [34, 35]) or allocate the whole pipeline for continuous execution until the client exits.

This flexibility in "when" and "where" to perform the allocation can improve the resource utilization efficiency for the potentially limited resources at the µDC.

3.8.1 Multiple data-flow graph instances per application and sharing

Objective-based resource allocation also forces the control plane to deploy multiple instances of the same application in different geographical regions for different subsets of clients. For example, there could be multiple autonomous cars driving around different AoIs in far apart locations. Additionally, the collaborative autonomous driving application has a tight latency requirement. Thus, each separate AoI could require an independent instance of the application deployed close by (network-wise) to the corresponding cars in the AoI. Similarly, standalone applications also require multiple instances deployed across the geo-distributed infrastructure. The tight latency staleness requirements of standalone applications restrict the processing of clients to a small subset of possible close by µDCs. The geo-distribution of the clients worldwide will cause multiple application instances to be deployed across spatially diverse µDCs.

One potential pitfall of this kind of design (where the control plane is deploying multiple DFG instances) is the unnecessary deployment of multiple application instances in the same µDC. In order to ameliorate the negative effect of multiple instances, the control plane can decide to share a certain application component instance in the DFG for a given application. Sharing components allows reusing the shared base memory footprint of applications across multiple clients hosted in the same set of µDCs. For example, if an application component uses a machine learning model to do inference, multiple clients can share the same application instance, and only one such model would need to be present in the memory of a server in the µDC, reducing the memory pressure overall.

Sharing application components across multiple clients and AoI reduces unnecessary waste of resources by reusing shared elements (like application binaries) and improving the utilization efficiency of resources (*e.g.*, cores) by multiplexing the same component across clients. The main complexity that arises is that the objective-based resource allocation needs to be aware that increasing the number of clients associated with a given instance could also require a different allocation of resources (*e.g.*, CPUs, memory). To solve this complexity, we later propose the use of offline profiling to calculate the required resources for a given number of clients in section 7.4.2.

3.8.2 Migration API

The use of objective-based resource allocation also requires that the control plane migrate application components due to either QoS or load-balancing considerations. The migration could potentially involve two main aspects: computation and state migration.

Computation Migration

Computation migration is related to migrating an application component instance from one μDC to another. To facilitate computation migration, the programming model provides the application with two handlers (Table 3.3), one to be executed at the current application component instance and the other at the new instance.

The first handler, *on migration start(child)*, allows the current application instance to package the volatile state of the computation of the parent node concerning the specific identifier (either of the AoI or client). The second handler, *on migration end(s)*, allows the new application component instance to initialize its local state using the transferred state for the corresponding identifier. The control plane will call the first handler on the current application component instance and ship the state to the new instance. Once the transfer is complete, it is safe to switch the logical partition to the new parent. The new parent node will start processing event calls from this transferred logical partition as soon as the initialization of its local state is complete. Finally, we discuss further optimizations to the data transfers in chapter 4.

State Migration

State migration relates to the soft persistent data generated by an application component (in the object store mentioned in section 3.5.3), which must be made available if the application component is migrated to a new μDC. The state migration can be performed lazily in parallel with execution in the new instance.

3.9 Limitations

As mentioned previously, the programming model presented in this chapter is a starting point for translating the requirements of situation-awareness applications into an intuitive interface that can be leveraged by the control plane. As such, it is not the final interface for all types of situation-awareness applications. This section discusses some of the main limitations that future iterations would require to address.

The first limitation is the composition and support for heterogeneous types of clients within one application. We assume that incoming sensor streams are logically presented for only one type of client—for example, drones for the standalone exemplar application and autonomous cars for the coordinated application. The dataflow graphs cannot easily represent the aggregation of different types of clients as the mechanisms for grouping clients of different types and their corresponding requirements are beyond this dissertation scope. Therefore, future extensions will be needed to group multiple independent standalone and coordinated applications for disparate types of clients, such that the dependency between different types of clients is easy to specify. This interface is discussed further in section 10.2.2.

The second limitation is that the current incarnation of the programming model does not include feedback loops that would facilitate multi-iteration processing. If the programming model is to be extended to domains beyond situation-awareness, wherein such facilities are needed then it would require additional features. For example, some stream-processing engines, such as Naiad [36], support using feedback edges to perform iterative computations, like those used for training machine learning models. The expectation is that the feedback loop data is treated differently than a request coming through the forward edges. The loop-back messages should not trigger computations that generate an output tuple in the node and should only be used for updating models and/or affecting the real world via actuations.

The third limitation is that spatial affinity cannot change at runtime, as it is defined as a static mapping function. In the future, some applications may require reconfiguring the AoI based on new knowledge, like, for example, the change in the importance of an area during the day or to support interactive augmented reality games where the gathering areas change with time. Such applications will require the programming model to represent how and when to trigger these reconfigurations.

Finally, the programming model does not provide any data-delivery guarantees. This design is intentional as situation-awareness applications mainly focus on quickly reacting to the physical world, where the data and reactions are only relevant for a limited time. Supporting delivery guarantees, such as at-least-once, would add to the latency overhead and hurt the application-level E2E latency SLOs. Therefore, we did not consider providing such guarantees in the current iteration of the programming model. However, including such guarantees would increase the versatility for future iterations of the programming model.

3.10 Conclusion

In this chapter, we first analyzed the structure and requirements of situation-awareness applications, and defined a taxonomy of those applications. Then, based on the insights obtained from this categorization, we proposed a programming model capable of defining geospatial requirements and providing guides to the control plane to better distributed and scale the application across the geo-distributed infrastructure. Additionally, we showed (via examples) how expressive this programming model can be for implementing situation-awareness applications. In the following chapters, we will build on the requirements presented before to define the required logical components in the control plane and define implementations and distributions of those components on the geo-distributed infrastructure.

CHAPTER 4
ARCHITECTURE FOR A CONTROL PLANE FOR GEO-DISTRIBUTED RESOURCES

The control plane is in charge of managing all the geo-distributed resources and the life cycle of the applications. It handles the scheduling of applications, monitors both resources and applications (including failures), and manages data related to the resources. This chapter discusses the requirements and challenges of a control plane that manages geo-distributed resources, and we describe an architecture that fulfills those needs. The chapter's main focus is on the functional and performance properties of the control plane —fault tolerance and reliability are outside of the scope of this dissertation.

4.1 Application life cycle overview

The life cycle of an application demonstrates the interactions between the runtime handlers and the control plane. There are three main groups of actions involving the control plane: registration, resource assignment, and reconfiguration via monitoring. Next, we present an enumeration of workflows for each of them.

Registration. The developer needs to register applications with the control plane as follows:

1. Each application is uploaded as a DFG by the application developer to the control plane, as previously explained in section 3.7.

2. Once registered, the client (*e.g.*, devices, sensors) will use the associated's application identifier to contact the control plane.

Resource assignment. The control plane defines a set of application components for a client (and potentially allocates them), with a common life cycle being as follows:

1. The client makes a request to the control plane with the associated application's identifier.

2. The control plane examines the state of the geo-distributed resources and selects a subset of those resources to schedule the application components, and deploys the required application components.

3. Once the application DFG is deployed in the corresponding µDCs/DCs, the client connects to the corresponding application component and keeps using it until it is no longer required or the control plane reconfigures the application.

Reconfiguration. The control plane may need to reconfigure certain already running application components and clients to maintain the required SLOs. A subset of these operations is shown next:

1. The control plane monitors the application components and resources to anticipate / detect any service-level agreement (SLA) violation or failure and react accordingly by reconfiguring (*e.g.*, migrating or restarting) some (or all) of the application components.

2. The control plane also periodically load balance applications across equivalent µDCs / DCs

The control plane automatically manages the discovery as part of resource assignment and migration of application components as part of reconfigurations. An example of a reconfiguration may happen when a client is mobile. The deployment for the client's application instance may need to be changed from one µDC to another as the location of the client changes.

4.2 Requirements

The properties of situation-awareness applications and those of geo-distributed resources (*i.e.*, µDCs and DCs) impose a set of requirements on the control plane that manages them. For example, situation-awareness applications are bursty in resource requirements across time (*e.g.*, rush hour traffic vs. early-morning traffic) and space (*e.g.*, cameras near busy intersections and those in secluded areas). Additionally, the resources in µDCs are potentially limited. Thus, the control plane will need to dynamically and continuously adapt the applications' resource allocations to efficiently use the available resource as they can not be defined statically.

Situation-awareness applications have distinctive requirements, beyond latency and bandwidth constraints, in contrast to well-known Cloud-native applications. These requirements are:

R1 Autonomous control: Each µDC/DC should be able to perform operations autonomously. Given the geo-distribution of µDC, partial disconnections would be more

common than for resources within a cloud datacenter. In order to improve availability, each µDC should be capable of managing the local resources, responding to any requests that are sent directly to it, and reconfiguring applications that are locally deployed. This requirement is especially important for *standalone* applications that tend to have tight latency bounds.

R2 Coordinated control: Multiple control plane actions will require a coordinated operation across multiple µDCs/DCs. Examples of such operations are migration of components across two µDC, load balancing across equivalent µDCs/DCs, and guaranteeing uniqueness of deployments for *coordinated* applications. These operations require that the control plane support decisions spanning multiple geo-distributed resources.

R3 Spatial knowledge: the control plane should understand the semantics of geographical location and the application's requirements to provide the correct functionality of applications. The control plane should consider each client's geographical location when making decisions, especially for coordinated applications. For example, some types of situation-awareness applications require sharing state among subgroups of clients based on their geographical location, as described in section 3.4.3.

R4 E2E latency support: the programming model in chapter 3 uses a DFG to model the topology and communication patterns of the application. In order to properly support E2E latency constraints, the control plane will need to be able to aggregate information across each application component (potentially deployed in different µDCs/DCs) and understand the application topology to support the necessary end-to-end latency SLO guarantees and their corresponding management. Additionally, the control plane will also need to have an approximation of inter-µDC latencies to calculate expected E2E latencies when deploying.

R5 Dynamic resource allocation: given the potentially limited resources in µDCs, and the continuously changing applications (*i.e.*, execution, latency, and location), the control plane will be required to perform dynamic resource allocation needed for the redeployment of an application due to mobility or failures/resource scarcity at an µDC, as well as dynamically modifying application component's resource allocation to better match is current load.

This set of functional requirements will guide the architecture of the control plane in the following sections of this dissertation. Additionally to the specific requirements specific to

the context of this dissertation, the control plane also has similar requirements to regular cloud control plane architectures, shown next:

Scalability: it should be capable of handling an increasing number of both devices and applications when more resources and µDCs are assigned to the control plane. This requirement also involves having simple operational management, such that scaling up/down is an easy operation.

Reliable/available: it should gracefully handle failures in its components and the network.

Agility: the control plane should respond to changes and requests quickly. Situation-awareness application workload and environment are continuously evolving; the control plane should reconfigure the application allocation and placement to provide the expected quality of service.

Maximize utilization: resources are limited; the control plane should maximize the utilization of the available resources. This requirement also involves minimizing any unnecessary resource (over-)allocation.

This second set of requirements extends the functional requirements of the first set to define performance requirements of the control plane for it to be practical to deploy situation-awareness applications in geo-distributed computational resources.

4.3 Challenges

New challenges arise in the design of control planes due to the geographical distribution of µDCs/DCs. The geo-distribution of resources will cause each µDC to be smaller than regular datacenters, as we can see from the Azure + AT&T project [15], as well as newer startups like VaporIO [17]. Smaller capacity means that fragmentation and over-allocation have a more significant impact on the QoS. Geo-distribution also means a higher likelihood of traversing the WAN when different control plane components are communicating over non-homogeneous and potentially high latency links.

Additionally, the requirements of situation-awareness applications can be quite stringent, more than current applications such as video streaming. For example, the missing child application (from section 2.2) requires both low-latency processing and a high-bandwidth network. The input device (*i.e.*, Google Glass) continuously generates data, and many IoT devices similarly generate data 24/7. Besides the applications generating a

significant amount of data, their context and workload are also constantly changing; the devices are moving around, and the number of elements that the sensors measure also varies with time.

Both the properties of geo-distributed resources and those of the situation-awareness applications increase the complexity of the control plane compared to those in cloud datacenters. In the following sections, we first explain why the current state of the art is not sufficient (section 4.4) to provide the requirements from section 4.2, and then in the remaining of this chapter, we propose a logical architecture that can provide each of these requirements.

4.4 Related Work

This section describes the current state of the art of control planes and resource managers and analyzes why they are unfit for geo-distributed resources. The two main deficiencies of state-of-the-art control planes when managing geo-distributed resources for situation-awareness applications are:

- They have a centralized architecture with WAN traversal needed for every action taken.
- Schedulers and policies are not cognizant (or not in the right granularity) of infrastructure topology, spatial affinity, and E2E latency.

Next, we analyze specific related work against the requirements of a control plane for this dissertation's domain.

Cloud control plane. Managing densely geo-distributed resources differs from conventional cluster scheduling. Cloud resource managers (*e.g.*, Borg [37], Omega [38], Hydra [6]) are designed for the features of a datacenter environment: mostly homogeneous computational resources interconnected by a low-latency network, which implies that the decision-making entity is colocated with the resources it manages. This coupling also manifests in how the allocation state is maintained in a centralized shared manner.

There is a large body of work in schedulers designed for Cloud datacenters, with the main difference across them being the structure of the data store and the schedulers' parallelism. This taxonomy separates the schedulers in monolithic [39, 10, 37], partitioned [6, 40, 41], hierarchical [42], and shared-state [38, 43, 44, 45] architectures. They mostly rely on a central authoritative state (which may be replicated for redundancy and fault tolerance) to coordinate resource scheduling, updated by one or more schedulers.

A shared authoritative state works in a datacenter environment for control plane decisions, but this approach is not scalable for edge-centric schedulers even when the schedulers are distributed, as in Omega [38], as they need to reach the centralized shared state over a high-latency and unreliable network for every control decision, which violates the requirements **R1** and **R5**. A naive extension of such schedulers to manage densely geo-distributed resources will not suffice to achieve all the requirements and is discussed in chapter 5.

Control planes for geo-distributed resources. Newer schedulers tailored for geo-distributed resources like KubeEdge [46] allow the µDC to work under network disconnection, but it does not allow new requests to be served directly at the µDC; requests are still processed in a centralized controller. These architectures suffer the same limitation as a cloud resource manager concerning requirement **R1**.

Additionally, neither cloud nor geo-distributed control planes are designed for the application requirements of situation-awareness applications. For example, they do not natively support location data (requirement **R3**), nor do they have the right monitoring capabilities to aggregate latency data across multiple application components and geo-distributed locations for providing requirement **R4**, as they rely on a centralized gathering of metrics. Additionally, they do not have the right interface to define how to trace latency across multiple application components in a DFG.

In summary, no current control plane architecture provides all the requirements.

4.5 Overview of the components in the control plane

The control plane is in charge of managing all the stages of an application life-cycle. The main operations performed by the control plane are:

Scheduling: it will choose the right subset of µDCs/DC that can provide the application requirements and have enough resources to host all the application components.

Managing application instances: it will be in charge of running each of the application components (*e.g.*, hosting the containers) and setting up all required communication channels between application components matching the topology of the DFG.

Managing application state: it implements the corresponding state API (section 3.5.3) and provides the best-effort access to the data independently of the location of the µDC/DC hosting a given application component.

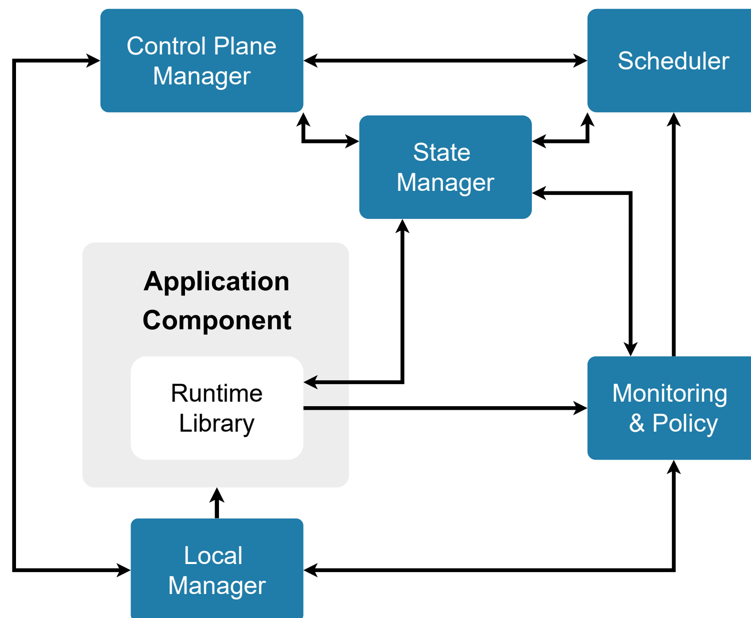

Figure 4.1: The control plane is composed of four main components: managers, scheduler, monitoring and policy, and state manager. The managers are further split into three main components: control plane manager, local manager, and runtime library. These logical components perform all the operations required by the control plane for managing geo-distributed resources and situation-awareness applications.

Monitoring applications and resources: it has to monitor the health of both applications and computational resources, such that it can redeploy any application component that has failed (either because of software or because of hardware). Additionally, the control plane needs to monitor each metric associated with the application requirements (section 3.4).

Policy execution: the control plane will use the output of the monitoring component and check (or predict) for potential violations of the application requirements. The policy will then trigger a rescheduling of some or all of the application components with additional restrictions to avoid using already known bad placements.

Each of these operations is performed by the independent components of the architecture shown in Figure 4.1, described in the following sections. This chapter focuses on the logical components required to implement a functional control plane for geo-distributed resources running situation-awareness applications. Later chapters will describe how these components are implemented, how and where they are hosted, and how these functionalities are further partitioned and replicated.

4.6 Scheduler

The scheduler is the entity in charge of selecting a subset of geo-distributed resources to host all the applications components for a given client (potentially reusing already running ones), and defining how many resources to allocate for each application component (*i.e.*, cores, memory). The scheduling functionality can be formalized as an optimization problem, which we will describe next.

The input to the scheduling optimization problem can be enumerated as follows:

1. Information about the client(s) to be scheduled: it includes the identifier of the application to be scheduled and the location of each of the clients.
2. The DFG that describes the application.
3. The application requirements (*e.g.*, E2E latency, spatial affinity) for each necessary application component.
4. The state of the geo-distributed infrastructure: the state includes the location of each of the µDCs/DCs, the current application components and allocations of each of them, and the total number of resources in each of those geo-distributed computational resources. Furthermore, for certain application requests (*e.g.*, with E2E latency requirements), the control plane will need the inter-site latency between µDCs.

The optimization will return for each application component either an already running instance or a specific µDC/DC to run the application and the number of resources to allocate to that application instance.

The optimization problem objective is to maximize/minimize the platform provider's metrics with the constraint of satisfying application requirements. It also provides a template for the provider to model the requirements of the infrastructure together with the needs of the application. The scheduling optimization can be optimally solved by an ILP solver (*e.g.*, Google OR-Tools [47]) or with a heuristic approach when the problem is too big or complex for an ILP to solve. We will discuss different heuristics in chapter 6 and chapter 7.

The platform provider defines the specific optimization metric to be used, so the metric definition needs to be extendable by the control plane manager for the control plane to be usable in a real deployment. All optimizations will still provide the application requirements, but from multiple potential solutions, it will choose the one that optimizes the

corresponding metric. Some examples of metrics that are relevant for a geo-distributed infrastructure are the following:

Cost: the optimization function will minimize the cost of the deployment while still providing the application's requirements.

Utilization: the optimizer will minimize the maximum utilization of the computational resources; this is a type of load balancing algorithm.

Imbalance: similar to utilization, it tries to improve the load balance of the infrastructure, but this metric will try to force the deviation between total allocation across different geo-distributed resources to be smaller.

Total applications: this metric optimizes for maximizing the number of applications that can be run in the infrastructure at the potential cost of increasing future violations (by reducing the margin of error to pack more applications).

Total clients: similar to the total applications, but at the granularity of the clients, this can affect the fairness across different applications.

Future violations: this metric would select the solution with the least likelihood of violations due to clients' mobility at the expense of more resource fragmentation or over-allocation of resources.

Statistical guarantees: in contrast to *future violations*, this metric allows a certain number of expected violations in the future to reduce the negative effects of over-allocation.

Fairness: a metric that balances the number of resources allocated to each application, such that each application receives roughly the same total number of resources.

Equation (4.1) exemplifies the optimization of the cost metric for an application with two application components in a pipeline with a data staleness latency constraint of 10 ms for the edge between the two application components.

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & \text{cost}(X) \\ \text{subject to} \quad & net(client, X_0) + computation(X_0) + net(X_0, X_1) \leq 10 \\ & placement(X_0) \neq \emptyset \\ & placement(X_1) \neq \emptyset \end{aligned} \tag{4.1}$$

The optimization in Equation (4.1) minimizes the placement cost, where the cost is calculated as the sum of renting the servers for both application components X_0 and X_1. Next, the net and computation functions calculate the expected latency when placing the two application components in X_0 and X_1 for the first and second components, respectively. Then, the constraint is to sum the three latency approximations and bound them to the limit of 10 (ms). Finally, the two last constraints define that both components should be placed in a valid μDC. The optimization metrics will generally vary considerably, but the base set of requirements will be similar across the different applications.

Many other potential metrics can be used to optimize the problem, and the definition of those metrics is outside the scope of this dissertation. Here, we presented a subset of the metrics to exemplify how a control plane can consider the metrics depending on a given infrastructure's specific context. The selection of a specific metric will heavily affect both the resource efficiency and the application's behavior. For example, fairness across applications may reduce the number of potential clients using the available resources at a time, given that some applications may require more resources. Moreover, it depends on whether the fairness is defined per μDC or overall resource in the platform, further exacerbating the problem. On the other hand, minimizing the number of potential violations could incur higher resource over-allocation, and reduce the overall resource efficiency, because it is allocating based on worse-case scenarios of both burstiness and potential mobility of users. Therefore, the infrastructure provider would ponder what metrics are more important for their business.

Another possibility for the optimization is to define a metric per application, but this would make proving efficiency much harder, given that different metrics could be at odds with each other, but that flexibility may be required to allow additional applications (similar to how the Linux scheduler has multiple schedulers for different applications).

The main aspects that affect how the scheduler optimizer will solve the problems that are different from cloud schedulers are:

- Geo-distribution and scarcity.
- Split across geo-distributed μDC and cloud DC.
- DFG semantics and reuse of components.

The geo-distribution and scarcity of resources reduce the equivalence between different computational resources making the search space bigger. For example, TetriSched [48], a cloud scheduler, uses equivalence sets to reduce the search space, given that many machines

can execute the same tasks and are deemed equivalent in the datacenter context. However, these types of optimizations are not as useful in the context of geo-distributed resources. Scarcity similarly forces the scheduler to consider partitioning the DFG across the cloud and edge to overcome the inherent limitations, as well as reusing components instances across different clients' DFGs.

Although outside of the scope of this dissertation, plenty of research is being done in improving the optimization of similar problems, which is discussed further in chapter 8.

4.7 Monitoring and Policy Definition

The monitoring component is the heartbeat of the control plane. It monitors several metrics associated with each application component. These metrics are then aggregated and used to detect whether that application instance needs to be reconfigured. The monitoring activity is not part of the application logic, but it is a necessary element of the control plane architecture (Figure 4.1) to ensure that the application's SLAs are met.

The monitoring component continuously gathers information pertaining to the SLOs of the application components and about each µDC/DC's resource usage. More specifically:

- Processing Latency.
- Network latency between connected components.
- Resource usage (*i.e.*, CPU and memory).
- Client's locations.

Since an application's DFG could span multiple µDCs, the measured data may have to be aggregated across µDCs and may need to be measured in different runtime components. Additionally, these measurements need to be further aggregated into metrics to be useful for decision-making.

To better understand the metrics being monitored, we categorize the type of metrics into *local-domain* and *global-domain*. For a given metric, if we need values from multiple components of an application DFG to decide whether a reconfiguration is required, we consider that metric as having a global-domain scope. For example, *processing* and *network latencies* have global-domain scope since values from multiple components of an application instance are required to determine if E2E latency constraints are violated. On the other hand, if the values for a particular metric need to be monitored only for a specific

application component to detect a need for reconfiguration, such a metric is categorized as having local-domain scope (*e.g.*, CPU usage). We use this categorization as a guide for improving the scalability of the monitoring layer in chapter 7.

The monitoring layer is also in charge of processing the metrics stream and applying application-specific policies. There are two main types of policies: proactive and reactive, depending on when actions are taken. Proactive policies involve taking actions either with periodic checks or by extrapolating the current system's state and predicting a violation. In contrast, reactive policies take action after a violation is detected. This taxonomy will allow us to explain the control plane implementations more easily in the following chapters.

There are at least three policies that are needed natively based on the control-plane requirements (section 4.2):

1. **Latency staleness:** The main policy is meeting the latency staleness SLA, with the associated latency metric being a global-domain metric. For example, if the measured E2E latency for a given application instance changes from its predicted value during runtime, it might trigger a proactive migration to avoid E2E latency constraint violation. Then, the policy will be fed back a reconfiguration request to the scheduler. The reconfiguration will request a replacement and could potentially indicate μDCs to avoid.

2. **Spatial affinity**: this policy is in charge of the spatial affinity that continuously analyzes each client's location and triggers corresponding migrations when moving to a different AoI.

3. **Load balancing:** the third policy load balances across equivalent μDC/DC resources. The monitoring layer periodically scans the entire infrastructure for significant load imbalances. The scheduler will be called to perform reallocation or migration decisions to ameliorate the problem when such incidents are detected.

In summary, the monitoring component supports an agile control plane through continuous measurements of the health of the μDC/DC combined with application-specified attributes.

4.8 Control Plane Managers and Runtime Library

The control plane manager is the entity in charge of executing actions decided by other components. For example, the control plane manager deploys the application components

in the locations defined by the scheduler and receives requests from clients. More specifically, the control manager performs the following actions:

- Receives and manages requests from clients and other components.
- Communicates messages between the different control plane components.
- Coordinates across components and between µDCs/DCs.
- Deploys the application components (*e.g.*, as containers) in the corresponding serving within a µDCs/DCs
- Interacts and monitors the physical hardware running the application components.
- Bootstraps all control plane components and monitors their health.

In order to fulfill all these tasks, the control plane manager requires hosting a software agent in one of the physical servers in each µDC in the infrastructure. The control plane will initially forward the initial requests to those machines, set up connections, and check those components' health. Additionally, these *local manager*s also help collect metrics from the applications for the monitoring layer (e.g., resource utilization of an application component).

In addition to the control plane managers and *local managers*, an additional component runs together with the application components: the runtime library. The runtime library implements the APIs used by the application components (section 3.5). In addition, it manages all communication requests and interacts closely with the control plane manager.

The runtime library is in charge of directly contacting the control plane for any scheduling request. In addition, once the application components are deployed, the runtime library is in charge of communicating between the client and the first application component, as well as between connected application components. The runtime library hides away all the complexity of geo-distribution from the developer and all coordination required across different application components in the system.

The runtime library will be included as a library of the binary that will execute the applications (either in the client or inside the containers hosting the application components). Additionally, since it runs as part of the same binary and schedules the execution of the event handlers presented in section 3.5, the runtime library also measures certain metrics of interest for the monitoring layer, like processing latency and network bandwidth utilization. The runtime library has full visibility of the execution of each event handler, as well

as all the communication between elements in the DFG. More details about its implementation are discussed in later sections.

In summary, both the control plane manager and runtime library are in charge of executing the actual actions specified by the other components of both the application and the control plane.

4.9 State Manager

The control plane is also in charge of managing, storing, and giving access to the state. More specifically, the state manager handles the state related to the infrastructure and each instance of the application components. The infrastructure state refers to the information related to the geo-distributed resources—for example, allocation, application components running, distribution, and application profiles. The scheduler and the policy executor (part of the monitoring component) need access to the infrastructure state to make decisions about applications in the system and deploy and reconfigure applications. On the other hand, the application state relates to all the data stored for a given application.

The application state is split into two parts: static and dynamic. The static application state is associated with the application component binaries and associated configurations and all the inputs submitted to the control plane by the application developer (section 3.7). The static application information can initially be stored in the cloud and replicated on-demand (or proactively) to the different required geo-distributed computational resources when they need to deploy the applications. The dynamic application state is associated with the dynamic state generated through the *data management API* (section 3.5.3). An implementation of the dynamic data management is presented in section 6.4.2.

One key aspect that the state manager of the control plane handles is the migration of data between two different application component instances for the same node in the DFG. Previously, we described that application components might be migrated (section 3.8.2) due to failure or potential violations of SLOs. The control plane is in charge of giving access to that data in a best-effort approach. This requirement may involve migrating data from a previous application component instance running in a different µDC/DC. Implementing this control plane architecture will need to define mechanisms to fetch the data and make it accessible to the corresponding application component and delete old data when it is not required. As part of the programming model, we do not expect the dynamic state management to be resilient to failures. The trade-off in this architectural design is to provide fast access to data, but all state is expected to be recomputed if the data is not found (either

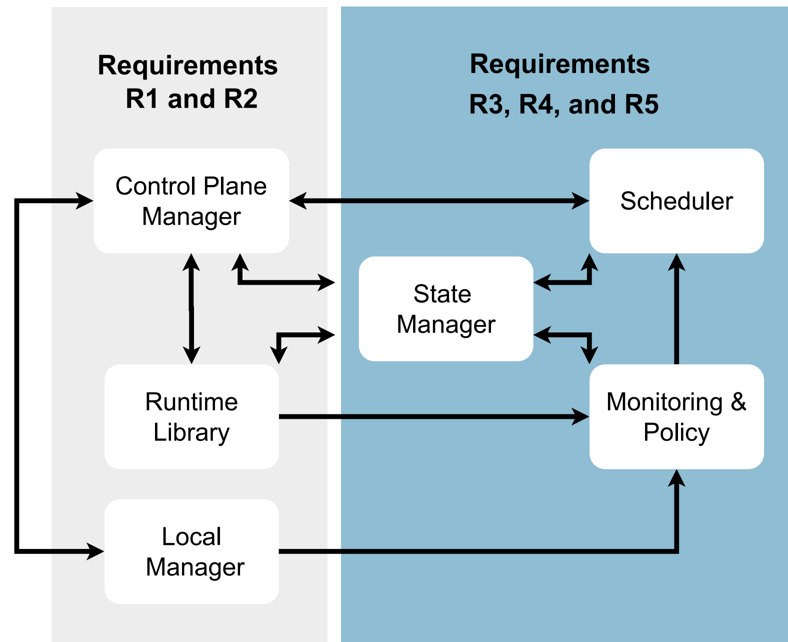

Figure 4.2: Relationship between functional requirements and logical components of the control plane architecture. The components highlighted in gray (left) focus on both control requirements **R1** and **R2**. The components highlighted in blue (right) are the main drivers behind providing dynamicity and supporting the situation-awareness application's special requirements.

because of failure or because it did not previously exist).

The state manager will also require additional logic in the runtime library presented in the previous section. The runtime library will implement the state management API section 3.5.3 in order for the state manager to decide where and how to save the data. The application state API avoids unnecessary recomputation and provides a helpful interface to query spatio-temporal data generated from the clients' inputs. In chapter 6, we present efficient mechanisms implementing this key-value interface.

4.10 Discussion of the architecture and requirements

Figure 4.2 shows the relationship between the functional requirements (section 4.2) and the control plane components in the architecture (section 4.5). Each component is involved in providing many of the functional requirements.

For both control requirements **R1** and **R2**, most of the functionality falls on the *control plane manager*; this is highlighted in gray (left) in Figure 4.2. Requirement **R1** is mostly

provided by the *local managers* in each of the μDCs/DCs. This design is due to the need for each μDC to be independent, which can only be achieved by having local entities in each μDC. On the other hand, requirement **R2** is achieved through the *control plane manager*, given that it is in charge of coordinating actions across different μDCs/DCs.

The other three requirements **R3** to **R5**, are centered around the monitoring component, as shown in blue in Figure 4.2. The monitoring and policy components continuously ingest the metric streams from different runtime and infrastructure components. This information is combined with the state of the infrastructure for the policy to detect potential violations or failures and then trigger new scheduling with the corresponding restrictions based on the root of the violation/failure. This stream processing is required given that the three requirements need verification at each moment in time. After a reconfiguration is deemed needed by the policy, the scheduler and the control plane manager will be required to decide and configure the infrastructure.

The performance requirements are discussed in the following chapters of this dissertation.

4.11 Distribution of control plane components

Given the described logical architecture, this dissertation's main research contribution determines how to partition each control plane component. This partitioning specifies where in the geo-distributed infrastructure to execute each component and what mechanisms are required for components to perform efficiently in a geo-distributed infrastructure when running situation-awareness applications.

The simplest (and less efficient) approach would be to perform most of these operations in a centralized location. On the other hand, in a geo-distributed architecture, the control plane can distribute the components among the distributed infrastructure. For example, we can implement a fully decentralized architecture, where each μDC/DC performs local scheduling independently and coordinates across nearby μDCs to handle operations like monitoring and migrations. In the decentralized design, the dynamic state management is handled independently by each μDC, and when a client is migrated to a different μDC, the state migration needs to be coordinated between the two locations. We analyze the decentralized architecture further in chapter 6.

There are additional strategies in the design continuum between fully centralized and fully decentralized for each of the components described in this section (*i.e.*, scheduling, monitoring, state management). An example is a hybrid design in which part of the oper-

ations are performed decentralized and others in a centralized fashion (within each of the control plane components). In this dissertation, we analyze in detail the effect of the distribution of components on how the control plane can cater to the requirements explained in section 4.2

4.11.1 Preview of following chapters

We will progressively implement a control plane in the following chapters that can provide all the requirements described in section 4.2. First, we analyze a fully-centralized design, where most of the logical components are deployed in a centralized manner, and show why it is not a good fit for the context of geo-distributed resources and situation-awareness applications. Next, we describe a decentralized design that can cater to E2E latency requirements. Finally, we propose a hybrid design that can satisfy all the requirements presented in section 4.2.

The main differences we focus on across the different designs are:

- The distribution of the control plane components, as previously mentioned in section 4.11.
- The implementation of the different mechanisms considering the distribution of components, including coordination, scheduling, and monitoring.

CHAPTER 5
ANALYSIS OF A CENTRALIZED ARCHITECTURE AND ITS LIMITATIONS

This chapter presents the limitations of extending a centralized cloud control plane to support the requirements in section 4.2. We use Kubernetes [49], a well-known open-source centralized control plane, as our starting point. First, we describe Kubernetes, focusing on its architectural design, state management, and how it handles scheduling decisions. Next, we show that extending Kubernetes to handle the requirements is inefficient through an architectural analysis and latency evaluations on a prototype. Finally, we also include an analysis of newer Kubernetes extensions for geo-distributed resources, which also suffer from similar limitations. Parts of this chapter's contents were previously presented in OneEdge [22].

5.1 Background: Kubernetes—a centralized control plane

Kubernetes [49] is an "open-source system for automating deployment, scaling, and management of containerized applications" [50]. It is based on the architecture of Google's internal orchestrator Borg [37], but re-implemented for broader adoption by the general technology community. Next, we present a general overview of its architecture and functionalities relevant to implementing a geo-distributed control plane.

5.1.1 Control plane architecture

A Kubernetes system is composed of two main elements [51]:

Worker nodes: A set of *servers* that directly run the application components (*i.e.*, containers). Each server has an associated *local manager*.

Control plane: A manager of worker nodes and the associated containers running in them that is in charge of making decisions for the overall infrastructure.

The main role of Kubernetes is to handle and run containerized applications. The smallest unit of computation managed by Kubernetes is a *pod*. A *pod* comprises one or more tightly coupled containers (application components) that share resources in the same host. In other words, a *pod* represents a logical host for an application [52]. A *pod* can share storage and network resources, and all are described with one specification (*i.e.*, a Yaml file)

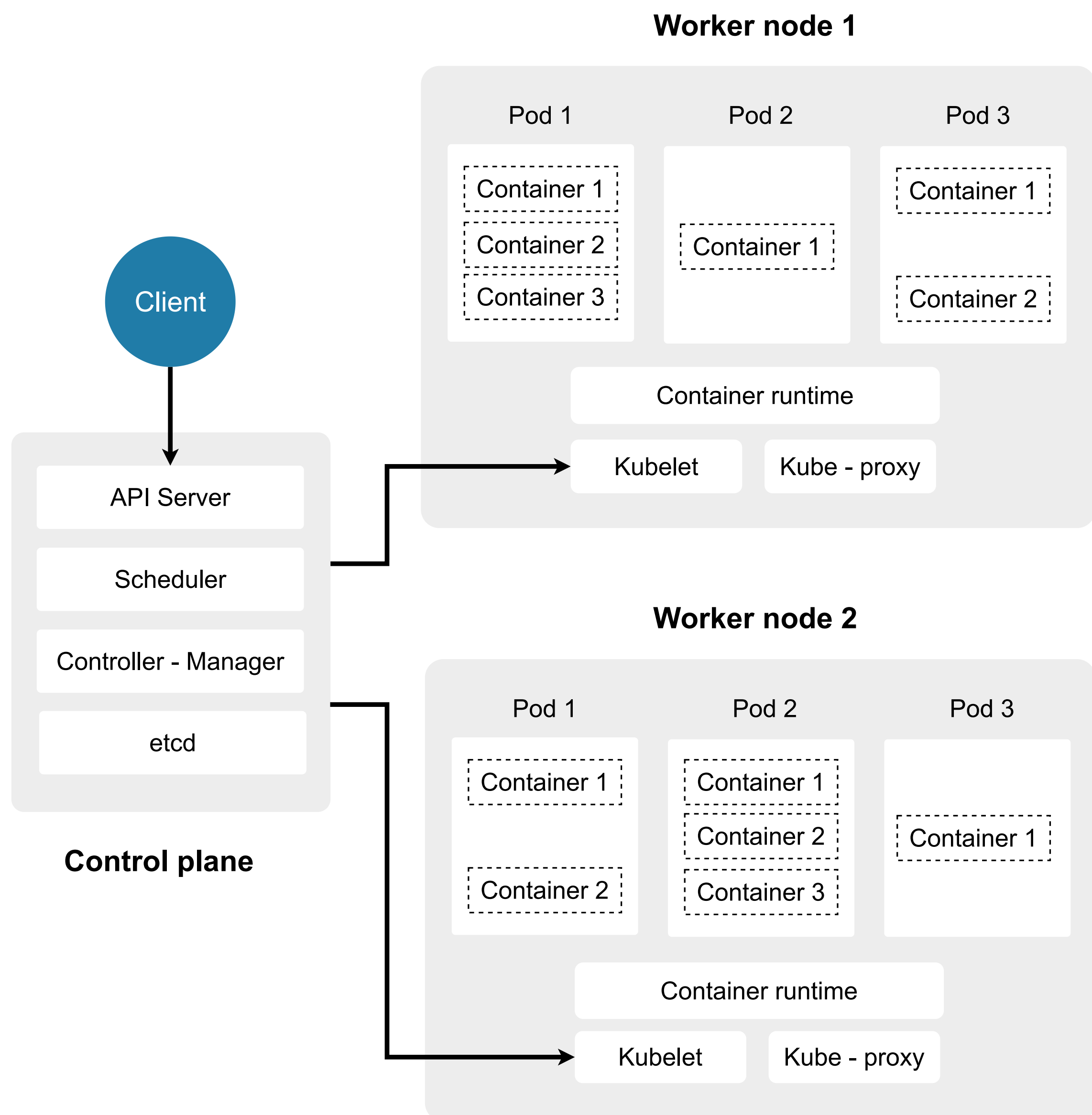

Figure 5.1: Architecture of Kubernetes. It comprises two main components: the *Control Plane* and the *Worker Nodes*. There is one logical control plane for the overall Kubernetes system and one instance of *Worker Nodes* for each server.

that indicates how they should be deployed. Pods resemble the DFG explained previously (section 3.3), but with the restrictions that all containers within a *pod* are *co-located* [52] within **one** server.

These high-level elements of the architecture are shown in Figure 5.1. Next, we describe each of these elements separately and compare them against the previously described architecture in section 4.5.

Worker nodes

Figure 5.1 shows the structure of a Kubernetes' *worker node*. Each worker node is composed of three main components:

Kubelet: it manages pods (and their constituent containers) and verifies that they are healthy.

Kube-proxy: it handles the networking rules associated with each pod. The rules allow communication between Pods in the cluster and external connections.

Container runtime: it is in charge of running the containers on top of the hardware.

In summary, these three components are run in each server and allow communication to the Kubernetes control plane, deploy the containers, and allow communication across components. Together they are functionally similar to the *local manager* in section 4.5.

Kubernetes control plane

As a clarification, in the context of this dissertation, we use a different definition of *control plane* than in Kubernetes. In other chapters, the terminology *control plane* would involve both what Kubernetes names worker nodes and control planes.

Four main components implement the control plane in Kubernetes:

Kube-scheduler: it processes any pod that is not assigned to a worker node and chooses a suitable node to execute.

Etcd: it is a key-value store used to store all infrastructure (cluster) and static application data.

Kube-apiserver: it is the front-end used by the clients of Kubernetes to send requests, including the deployment of Pods.

Kube-controller-manager: it runs all the controllers that manage the infrastructure (cluster) and pods. Kubernetes uses independent controllers for different types of components in the system. For example, it has separate controllers dedicated to managing and checking the health of worker nodes and workloads (*i.e.*, *pods*).

We can map these components to the control plane components presented in section 4.5:

- The *kube-scheduler* is directly associated with the *scheduler* and fulfills the same objective.
- *Etcd* implements the role of the infrastructure state manager and application state data, but it does not handle the dynamic application data. Application data is expected to be handled by other services running on top of Kubernetes.
- The *kube-apiserver* and the *kube-controller-manager* perform a similar role to the control plane manager, as they are in charge of coordinating operations, managing resources, and receiving requests from the other systems.

In addition to the core components of Kubernetes described before, there is one add-on to Kubernetes that is relevant for our discussion: the *container resource monitor* that maintains a time-series database with metrics related to containers and implements a subset of the functionality of the monitoring component in section 4.5. However, it does not apply policies on top of the metrics collected. Most of the metrics in this Kubernetes component are for after-the-fact analysis and not for real-time processing.

5.1.2 State in etcd

Etcd holds all the state of the control plane related to the cluster and the static data of the application. The state is persistently stored as objects called *Kubernetes objects* [53]. Each *Kubernetes object* describes the desired state of the associated object, or what in the Kubernetes documentation is called a ”record of intent” [53]. Once a given object is created, Kubernetes will try to keep that object in the desired state (including its creation). In other words, the creation of an object in *etcd* indicates to Kubernetes the need to keep that object in a specific state forever (or until the object is deleted or modified).

Most Kubernetes objects are composed of two fields that describe the object's configuration: the spec and its status. The spec describes its *desired state*, including the characteristic that the resource should have; the spec is not the same for all object types. On the

other hand, the status shows the real state of the object (continuously updated by Kubernetes' components).

The Kubernetes control plane continuously checks that every object's current state matches the desired state defined by the spec. For Kubernetes to implement this check, it leverages the *watch* feature from *etcd* [54]. An *etcd watch* provides an interface to monitor changes to specific keys. *Etcd* notifies asynchronously when a key is changed, similar to an event-based architecture. Kubernetes depends on these notifications to trigger an object's modification to match its spec.

The main information stored in etcd is:

- Information related to the containers (application components) running, including on which nodes and with what allocated resources.
- The resources that are available in each worker node.
- The spec that defines how each application behaves (spec). For example, restart and fault-tolerance policies and the number of resources required by the containers.

More specifically, two types of Kubernetes objects are important for this chapter discussion: workload and cluster [55]. *Workloads* are the objects used to handle the execution of Pods (and the associated containers) in the infrastructure. *Cluster* objects are associated with the nodes themselves and how they are configured.

5.1.3 Kubernetes API and workflow

The Kubernetes' clients (including other Kubernetes components) use the *Kubernetes API* to interact with objects (*i.e.*, creation, modification, or deletion). For example, the client submits a new spec of the desired object (*e.g.*, workload) or the corresponding change to a specific object. The Kubernetes API then validates the request and persists the associated information on etcd, which is only accessible through the Kubernetes API.

To better understand the behavior of Kubernetes, we enumerate the steps taken by the different components when a client sends an application deploy request, which is also depicted in Figure 5.2:

1. The client sends a request to the API server.
2. The API server validates the request and stores the associated state to etcd (with a corresponding acknowledgment by etcd).

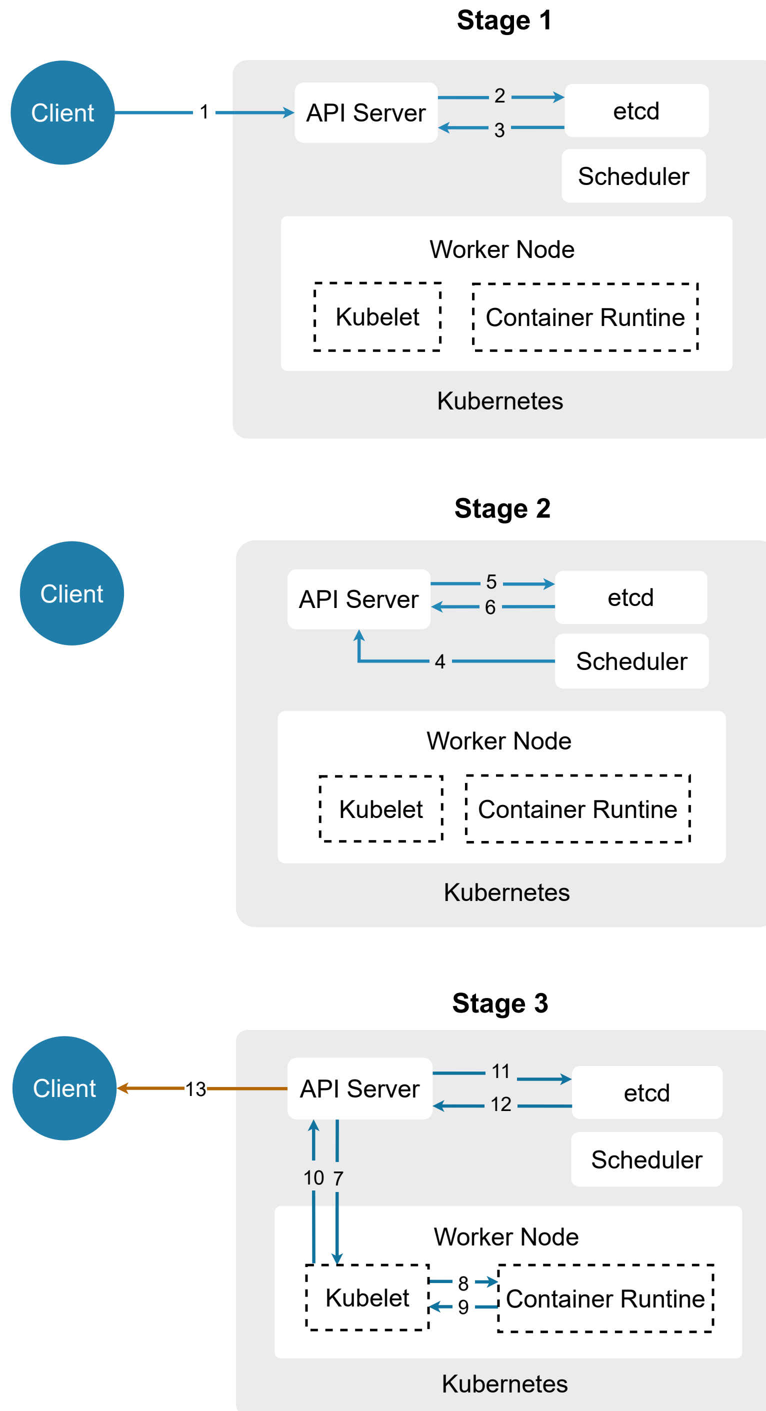

Figure 5.2: Three stages of the lifecycle of a *pod* creation in Kubernetes. First, the client submits the request to the API server, and the request is written to etcd. Then, in the second stage, a *watch* is triggered, the scheduler finds a suitable worker to run the pod, and the selection is saved to etcd. Next, in the third stage, the API server triggers the corresponding Kubelet, which deploys the application locally in the server using the container runtime. Finally, the result is saved to etcd when completed successfully (brown arrow).

3. The scheduler has a *watch* installed for changes on the pods. Then, the API Server notifies the scheduler about the new pod, which has no node assigned to it.

4. The scheduler decides which node to run the pod on and returns it to the API Server (if it exists).

5. The API server persists the decision to etcd (with the corresponding acknowledgment back from etcd).

6. Each Kubelet is watching for nodes assigned to them (through etcd), so the API Server notifies the corresponding Kubelet about the new pod.

7. The Kubelet requests the container runtime to start the corresponding containers.

8. Kubelet updates the pod status to the API Server.

9. API Server persists the new state in etcd.

After the initial deployment, the controller monitors the *pod* status (through the Kubernetes API), and when the pod status does not match the spec, the controller takes the required actions to move it towards the *desired* state. For certain types of faults, even the Kubelet can restart the container.

One key aspect to notice is that Pods are considered ephemeral and are scheduled only once. Once a Pod fails, in general, it will not be restarted, and the corresponding object is deleted from *etcd*. In order to provide fault tolerance, applications use higher-level object constructs, like *deployments*. The *deployment* object can represent the logical application running on the cluster. The spec of a *deployment* specifies the application's requirements, such as the number of replicas and the required hardware capabilities. The *workload* controller transforms a deployment spec into one or more pods. If a pod dies, the manager creates a new pod to substitute it. The corresponding controller will fix any other deviation of the *pod state* from the expected *spec*. *Etcd* maintains the current known state of a pod.

5.1.4 Scheduling

In the workflow presented in section 5.1.3, the Kubernetes scheduler is the entity assigning *pods* to worker nodes [56]. The scheduling process for a pod (also called the scheduling context [57]) is composed of two parts: the scheduling cycle and the binding cycle. First, the scheduling cycle chooses a node for the pod. Then, the binding cycle performs the actual deployment of the pod in the selected node. An important characteristic is that

the scheduling cycles are performed sequentially, while multiple binding cycles can be performed concurrently.

More specifically, the scheduling cycle steps [57] are as follows:

1. When a pod is added through the *Kubernetes API*, the scheduler is notified and adds the pod to its internal scheduling queue.

2. The scheduler picks a pod from the scheduling queue. The scheduling queue can be sorted before the next pod is chosen.

3. The scheduler filters the available worker nodes (and their resources) to match the pod's requirements.

4. The scheduler scores the remaining filtered resources, normalize those scores, and ranks them.

5. The scheduler picks a node with the highest score and reserves resources in it.

The scheduler determines which nodes can support the pod according to its constraints and available resources in the filter stage. If none of the nodes are suitable, the pod remains unscheduled until the scheduler can place it. Next, the scheduler ranks each feasible node in the scoring stage, and it picks a node with the highest score among the feasible ones. Then, it reserves the required resource in the node for the corresponding pod; it does this by saving the new state to etcd (through the *Kubernetes API server*). If an application is composed of multiple pods, each pod is independently scheduled and added separately to the scheduling queue.

For every newly created pod (or other unscheduled pods), the scheduler selects an optimal worker node for them to be executed. Every container in pods can have different requirements for resources, and every pod may also have different requirements. Therefore, existing worker nodes are filtered according to specific scheduling requirements. As part of this filtering, the main factors that are taken into account are:

Resource requirements: Currently, the only resource types supported in Kubernetes are CPU, memory, and hugepages [58]. These requirements are defined as the minimum number or size needed in the worker node to run the container.

Limits on worker nodes: For example, the maximum number of containers/pods that can be run in the node.

Labels: Constraints on a pod restricting it to run on a particular set of node(s). First, labels are assigned to nodes (*e.g.*, a location or a descriptive name). Then, labels selectors are added as part of the spec of the workload/pod. The scheduler can only choose nodes with the associated label.

Affinity/anti-affinity: An extension of labels, used in a more granular way via matching rules (besides exact matching) that can be created with logical operations. [59]. For example, we can use a rule to select any node that *does not* match a label. Soft-preferences are also allowed, such as ignoring a particular rule if no feasible nodes are available.

Inter-pod affinity/anti-affinity: Constraints on which nodes a pod can be scheduled based on labels of pods already present on the node. These constraints allow collocating certain pods together in the same set of nodes. For example, collocation can be used to improve data locality. Similarly, we can avoid interference between pods with anti-affinity labels.

Taints (applied to nodes): Allow repelling a set of pods from running in certain nodes.

Topology Spread Constraint: Defines how to allocate replicas across the cluster topology. It is a load balancing mechanism that allocates the same number of replicates across the different available labels for a given workload/application.

The scheduler treats all resources in the infrastructure as fungible and equivalent, except for the labels constraints and the available resources in the µDCs/DCs.

5.2 Designing a geo-distributed control plane with Kubernetes for situation-awareness applications

State-of-the-art control planes like Kubernetes do not meet the requirements (section 4.2) of geo-distributed resources and situation-awareness applications because they were designed for throughput-oriented micro-services running in the Cloud. As mentioned earlier, the requirements for such applications, except for requirement **R5** (dynamic resource allocation), are either not pertinent or easily met in a cloud environment. Support for fine-grained latency-sensitive or location-sensitive application placement needs to be built on top of Kubernetes as additional components. Furthermore, as we see in the following sections, they have a penalty on architectural complexity and performance.

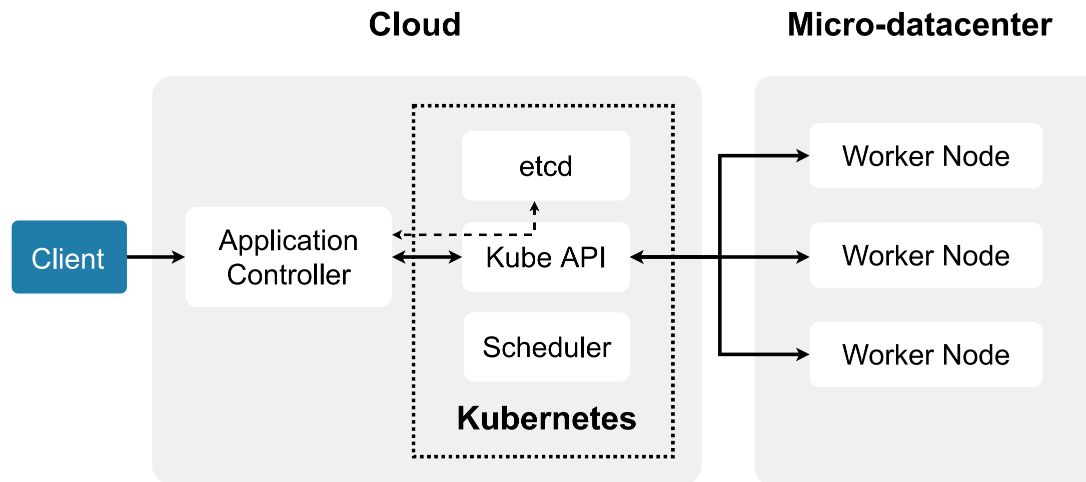

Figure 5.3: Extension of Kubernetes to support spatial and end-to-end latency resource scheduling and reconfiguration. An additional application controller is added to handle the semantics of these two deployment requirements, as well as to support atomic dataflow-graph deployments.

5.2.1 Enhancing Kubernetes to support spatial and end-to-end deployments

Now, we analyze a potential design that could implement the latency-sensitive and location-sensitive requirements on top of Kubernetes. First, an additional application controller must be added in the Cloud (Figure 5.3) to act as a layer above Kubernetes to instruct the scheduler on its desired placement decisions. The external application controller is required because Kubernetes has no notion of E2E latency or the geographical location of neither computational resources nor clients. The closest to supporting these requirements is labels, but the Kubernetes scheduler does not have the right abstraction to transform those labels to E2E latency and calculate latency across multiple pods (*i.e.*, application components) inside the *kube-scheduler*. Additionally, the deployment of a DFG needs to be atomic to provide all its functionalities (*i.e.*, all application components need to be deployed simultaneously), and the Kubernetes' per-pod deployment interface makes ensuring E2E end-to-end latency harder.

As a prototype, to provide both requirements **R3** and **R4** (latency and spatial-affinity), we add the following functionalities on top of Kubernetes:

- We add additional metadata and labels to each of the worker nodes in the system. The label associates a worker node to a µDC where it is located. The new metadata includes the geographical location of the µDC.

- All requests from clients are forwarded to the application controller instead of the Kubernetes API.
- The application controller has an eventually consistent local replica of the object in etcd, updated via watches. The replicated data involves mostly the deployed application components and the available resources in the different worker nodes. The replicated data is stored in a database that supports geo-spatial queries [60].

On a deployment request, the application controller selects the most viable µDC for each of the application components in the DFG, by reading the data in the replicated database and filtering resources based on all the application requirements (*i.e.*, resources, E2E latency, spatial affinity). Then, it requests Kubernetes to schedule each component in the DFG sequentially from the selected µDC. If any of those fail, the application controller must delete the previously associated pods/workloads from the Kubernetes API and restart the scheduling from scratch. The main functionality still used from Kubernetes is that the application controller does not need to choose which cores or servers in a µDC to use; we use the µDC-identifying label.

Additionally, since Kubernetes cannot implement a spatial and latency-aware controller, all monitoring data measured by the runtime library in the application components must be forwarded to the application controller to support all the required reconfigurations (*e.g.*, due to mobility).

These additional components provide the functionality of spatial and latency-aware deployments and reconfiguration on top of Kubernetes.

5.2.2 Distribution of components in a centralized architecture and workflow

The scheduling, monitoring, policy execution, coordination, and data management would be hosted in the Cloud as part of both Kubernetes and the application controller. The servers in each µDC/DC only have the Kubelet that is in charge of executing the commands defined from the central control plane. The Kubelet is unchanged with respect to Kubernetes and still connects through the *Kubernetes API server*. Additionally, all monitored data is forwarded to the central application manager.

Figure 5.4 shows the workflow of requests in this specific control plane implementation, similar to section 4.1.

①: The application's client sends an application deployment request to the dedicated application controller in the Cloud.

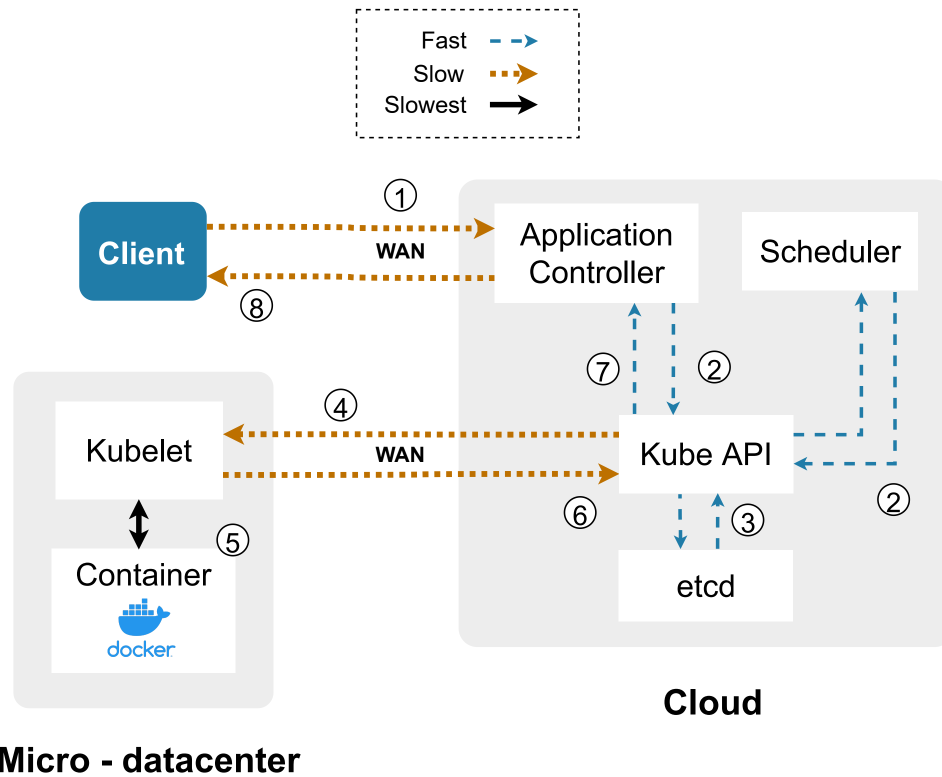

Figure 5.4: Workflow for application deployment in a micro-datacenter using Kubernetes.

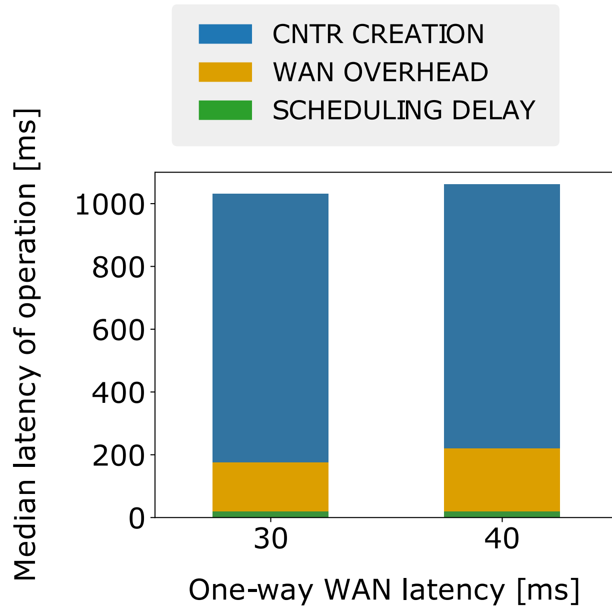

Figure 5.5: Experimental evaluation of an extended Kubernetes to support spatial and E2E latency requirements. The latency breakdown includes container cold start.

②: The application controller decides on a µDC/DC for each of the application components. The application controller sends sequentially scheduling requests for each application component to Kube-sched.

③: Once Kube-sched decides the specific resource for each application component, it enters the selection into the *etcd*.

④ to ⑥: The µDC/DC picks up the scheduling request, launches the needed application containers, and informs Kubernetes.

⑦ and ⑧: The application controller is informed that the application is ready to be used and notifies the client, which can now start interacting with the deployed application on the µDC/DC.

5.3 Limitations of a centralized design for situation-awareness applications and geo-distributed infrastructure

To understand the current state-of-the-art deployment overhead for situation-awareness applications, we conduct the following experiment with the extended Kubernetes prototype. In Figure 5.4, we describe the experimental setup used: a client of a situation-awareness application, the desired µDC for launching the application for the client, and the Kubernetes

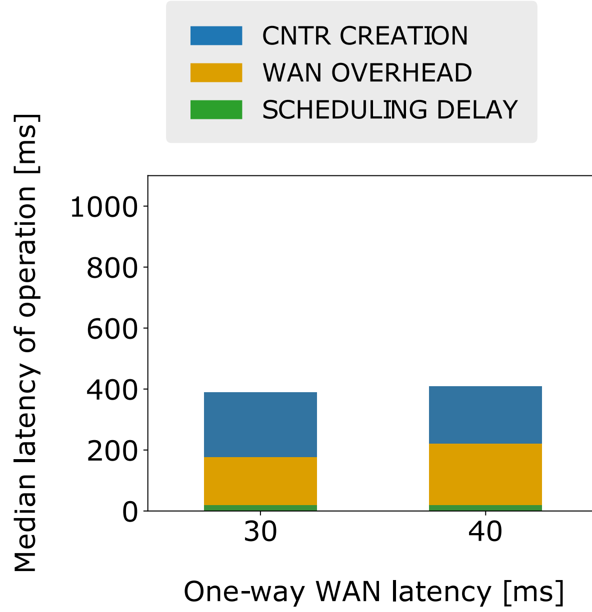

Figure 5.6: Experimental evaluation of extended Kubernetes to support spatial and E2E latency requirements. Latency breakdown when using a warm container.

scheduler and the application controller in the Cloud. The color of the arrows highlights the difference in latencies for each of the control plane operations. The control plane operations for deploying the application in a µDC are the same as in section 5.2.2. We focus on an application that only has one application component in the DFG (a best-case scenario).

We emulate all the entities (Client, Cloud, and µDC) involved in the control flow shown in Figure 5.4 as individual virtual machines (VMs) inside an Azure region [2]. This latency emulation and separation help showcase the best-case deployment latency (*i.e.*, no queuing effects due to other requests pending at the scheduler) for the extended Kubernetes prototype and different controlled settings of WAN latency. Every WAN hop shown in Figure 5.4 incurs a set latency controlled through the Linux *tc* [61] utility.

Figure 5.5 and Figure 5.6 show the average E2E deployment latency for two WAN latency configurations. Figure 5.5 depicts the latency when deploying containers anew (*i.e.*, cold-start), while Figure 5.6 is for pre-warmed containers. Each bar graph exhibits the latency breakdown into its components. The container startup time dominates the E2E application deployment latency for cold-start deployments, as shown in Figure 5.5. However, significant ongoing research efforts are tackling the high cost of cold start, like keeping pools of pre-warmed containers [62] to avoid this overhead. Then, once the cold start effect is reduced, Figure 5.6 underscores that by using pre-warmed containers, the overhead

of WAN traversal becomes the primary deployment latency component. For example, with pre-warmed containers and an 80ms round-trip WAN latency, the WAN overhead is about 49% of the deployment latency.

Ensuring low-latency control plane actions is essential for situation awareness applications both to get the application started initially and for reconfiguration decisions in response to client mobility or resource scarcity. Low latency is even more important when the control plane performs allocation decisions per message for a given application (ala serverless). If not executed quickly, control plane actions can result in E2E SLO violations for the applications, given that executions can be in the same order of magnitude as WAN latencies.

5.3.1 Geo-distributed Kubernetes extensions

This section presents two extensions made to Kubernetes to support geo-distributed computational resources better: KubeEdge and KubeFed.

KubeEdge

KubeEdge [46] is tailored towards geo-distributed edge computing devices. The main difference between Kubernetes and KubeEdge is the separation of the Kubernetes control plane into two domains: *CloudCore* and *EdgeCore*. The *CloudCore* domain contains the components of the control plane that run in a centralized cloud, while the *EdgeCore* domain runs in each μDC. Some components, like the infrastructure state and certain controllers in Kubernetes, are replicated across the two domains. For example, instead of the Kubelet in each server connecting directly to the Cloud through the *Kubernetes API server*, the data (associated with that worker node) from etcd is replicated in an eventually consistent manner to a local database in the EdgeCore domain. Additionally, KubeEdge also supports edge-centric features such as event buses connecting to well-known protocols used by edge devices and handling those sensors and devices.

This split between Edge and Cloud allows the worker node to keep functioning even when there is a disconnection from the Cloud: all the data is local, and the controller in charge of those objects is also local to the *EdgeCore*. Similarly, this split allows the controller of nodes (the replica in the Cloud) to avoid redeploying an application in a different worker node when there is a network disconnection. Instead, this controller waits until there is connectivity to learn about the current status of that object. In other words, the EdgeCore is now in charge of verifying the health of pods/workloads instead of the CloudCore in the

remote Cloud.

Even with these changes, KubeEdge still does not support autonomous control operations (requirement **R1**), but at least reduces the effect of disconnection. Furthermore, all limitations concerning placement requirements **R3** and **R4** persist in KubeEdge, as well as a lack of monitoring of spatial and E2E latency metrics. Finally, the latency for executing control plane operations (*i.e.*, deployments) still involves at least two WAN roundtrips. To verify the response time, we evaluated KubeEdge in the same setting as Figure 5.4 by using KubeEdge instead of Kubernetes. In this evaluation, we found that KubeEdge has a similar latency overhead to our extended Kubernetes evaluation (shown in Figure 5.6) but exhibits higher *overall* latency and variance due to additional book-keeping and state replication. Although KubeEdge uses a deployment workflow similar to Kubernetes, it suffers from worse performance due to these two additional operations.

KubeFed

The second project of interest is KubeFed [63]. KubeFed is focused on managing multiple clusters from the same interface. For example, we could consider each µDC/DC a Kubernetes cluster. KubeFed allows managing all of them from one unique interface instead of having to reach the control plane of each independent µDC/DC.

KubeFed uses a *host cluster* [64] (one of the clusters being managed) that exposes the multi-cluster *Kubernetes API* and hosts the control plane for KubeFed. The application configuration that will run across multiple clusters can be applied directly through the host cluster, which propagates the required specs and objects to all the corresponding clusters. The *host cluster* also allows a global view of all the clusters, which means that the state of the other *member clusters* needs to be replicated to the *host cluster* when there are queries to the *KubeFed Kubernetes API* (*i.e.*, it fetches the status of resources assigned by KubeFed across the member clusters).

KubeFed reuses the same type of objects as Kubernetes but creates a wrapper around them with *custom resource definitions*. In addition, the wrapper includes two additional multi-cluster variables [63]:

Placement: Selects the clusters where the application will be executed.

Overrides: Allows modifying the spec template specifically for a given cluster (µDC/DC). For example, overriding the replication to deploy in a specific server.

KubeFed suffers from the same limitations as KubeEdge and Kubernetes. Federated

deployments need to go through the *host cluster*, increasing the latency of federated operations, similar to section 5.3. KubeFed does allow local operations in each cluster autonomously, but for those resources created locally in a cluster, they are not propagated, and actions taken for those resources are only performed within that cluster. This limitation hinders its use when migrations are required due to E2E latency requirements as an additional coordination mechanism would be needed across clusters (μDCs/DCs).

Still, neither of them support neither spatial affinity nor E2E latency requirements, and additional components are needed to support the designs. The *placement* variable in the spec slightly simplifies the use of labels presented in section 5.2.1, but the limitations are still the same, and an external controller and replication of data are still needed.

5.3.2 Discussion of Kubernetes limitations

Besides the concern of placing multiple WAN traversals on the critical path of application deployments, there are three main limitations of Kubernetes from the mechanisms and architecture design perspective: the abstraction used to specify requirements, the scheduler architecture and assumptions, and the centralization of decision-making.

Kubernetes (and other state-of-the-art) schedulers do not natively cater to the requirements of situation awareness applications (section 4.2) in terms of meeting their E2E latency SLOs and respecting spatial affinity considerations. With the design in section 5.2.1, we provide a partial solution for the two requirements **R3** and **R4** (latency and spatial-affinity). However, it becomes a burden on the developer to build an application controller for each situation awareness application, given that there is no framework to manage this in Kubernetes across independent applications natively. Additionally, this incurs the overhead of completely replicating the state from etcd into an external database that supports geo-spatial queries. Also, its design does not handle dynamic application data or its migration. In order to support dynamic application data, the control plane needs to be aware of geo-spatial data of both the application component and the clients, and Kubernetes does not have support for this type of metrics and monitoring.

Another Kubernetes limitation that heavily impacts the support of situation-awareness applications is the design and assumptions of the scheduling algorithm. The Kubernetes scheduler lack support for scheduling multiple pods simultaneously (*i.e.*, gang scheduling). A multi-pod application will be deployed sequentially, which forces end-to-end constraints to be complex to deploy, a problem that grows worse the bigger the DFG is. The sequential deployment also causes a higher likelihood of deployment's failure due to divergences

from the eventually consistent data in the geo-spatial database and etcd, and the interaction across potentially multiple application controllers. The Kubernetes scheduler also assumes that many of the computational resources are fungible and equivalent, given that there will be many similar servers distributed among the datacenter. This assumption breaks in a geo-distributed setting with limited servers in each µDC; only servers in the same µDC are equivalent, and without good spatial support, the filtering and sorting algorithm are not efficient in selecting the right resources. Similarly, a problematic assumption is that pods are ephemeral; this limits the way soft-data can be handled by the control plane, given that at any moment, a container could be killed and restarted as a way to implement scale-up changes instead of adding resources in place (currently supported by the container runtimes), which causes the in-memory data to be lost.

Finally, the components' centralization (application controller and Kubernetes' components) forces all requests to be forwarded to the Cloud, violating the requirement **R1** (*i.e.*, autonomous control). Likewise, the reliance on a not-partitionable centralized state in etcd, and how the watches are all created from a central manager for nodes, is a big architectural limitation of this design that makes the support of requirement **R1** hard. This limitation is not yet solved even in geo-distributed versions of Kubernetes (*i.e.*, KubeEdge and KubeFed). The centralization of the components also forces a huge amount of monitoring data to be forwarded. Availability is affected because all Kubelets are dependent on the centralized Kubernetes control plane layer and cannot make autonomous decisions under a disconnection (which has a higher probability in WAN settings). Similarly, the Kubelet cannot handle most types of application component reconfigurations. Even with a design like KubeFed, all federated decisions need to be performed in a centralized manner.

5.4 Chapter summary

Kubernetes is an open-source centralized control plane for managing containerized applications. Even though it can handle the high-level operations of scheduling containers in a geo-distributed setting, it has serious performance and functional limitations, and, as noted above, its existing mechanisms do not inherently cater to the requirements (section 4.2) of a geo-distributed control plane running situation-awareness applications.

Its centralized design is a major limitation when the geo-distributed infrastructure is connected through WAN, as it does not allow autonomous control and forces all data and requests to be sent to the Cloud with a higher response time and bandwidth overhead. More importantly, the lack of native support for spatial and E2E latency metrics and scheduling

causes both functional and performance issues that limit the dynamic and fast response to application allocation needs, as any solution either depends on major duplication of efforts or big architectural changes that do not match Kubernets original goal. Therefore, we take a clean-slate approach to systematically address all of the requirements in the remaining parts of this dissertation while leveraging some of the concepts presented in this chapter.

CHAPTER 6
DECENTRALIZED ARCHITECTURE

The previous chapter analyzed a centralized control plane architecture and how it was unfit for geo-distributed resources and situation-awareness applications. One of the main limitations is that the µDCs are not close to the entities that manage them and make the scheduling decisions. This chapter presents *Foglets*, a fully decentralized control plane architecture. This architecture tackles the requirements **R1**, **R2**, **R4** and **R5** (*i.e.*, autonomous and coordinated control, dynamic resource allocation, and E2E latency support) of geo-distributed resources and situation-awareness applications. We present the implementation of the components in section 4.5 and how they are distributed in a geo-distributed infrastructure. This chapter focuses on the mechanisms and algorithms required by the control plane to implement the required functionalities in a decentralized fashion. The content of this chapter is extracted from our previous work, Foglets. [23].

6.1 Architecture overview and distribution

The main components of *Foglets* are shown in Figure 6.1. All components of *Foglets* are geo-distributed at different geographical scales. The main control plane actions (scheduling, management of both application instances and state, monitoring, and policy execution) are performed in a fully decentralized manner by the *local manager* running in each µDC, with some coordination across nearby µDCs/DCs. The *local manager* service runs in one of the servers in the µDC/DC and awaits requests from the application clients. The *local manager* also manages the servers in a specific µDC/DC. It implements the admission policy and the deployment of applications. Additionally, it provides other functionalities discussed in further sections of this chapter.

The discovery service is a partitioned name server that lists all µDCs available for hosting application components for a given geographical area. Each *local manager* periodically pings the discovery service as a health check and keeps the list of available resources up-to-date. The µDCs that are most useful to a client are the ones that are in geographical or network proximity. *Foglets* leverages this intuition to geographically partition the discovery service with an eventually replicated copy of other further away µDCs/DCs (from other discovery service instances). Additionally, the client to the dynamic service can also

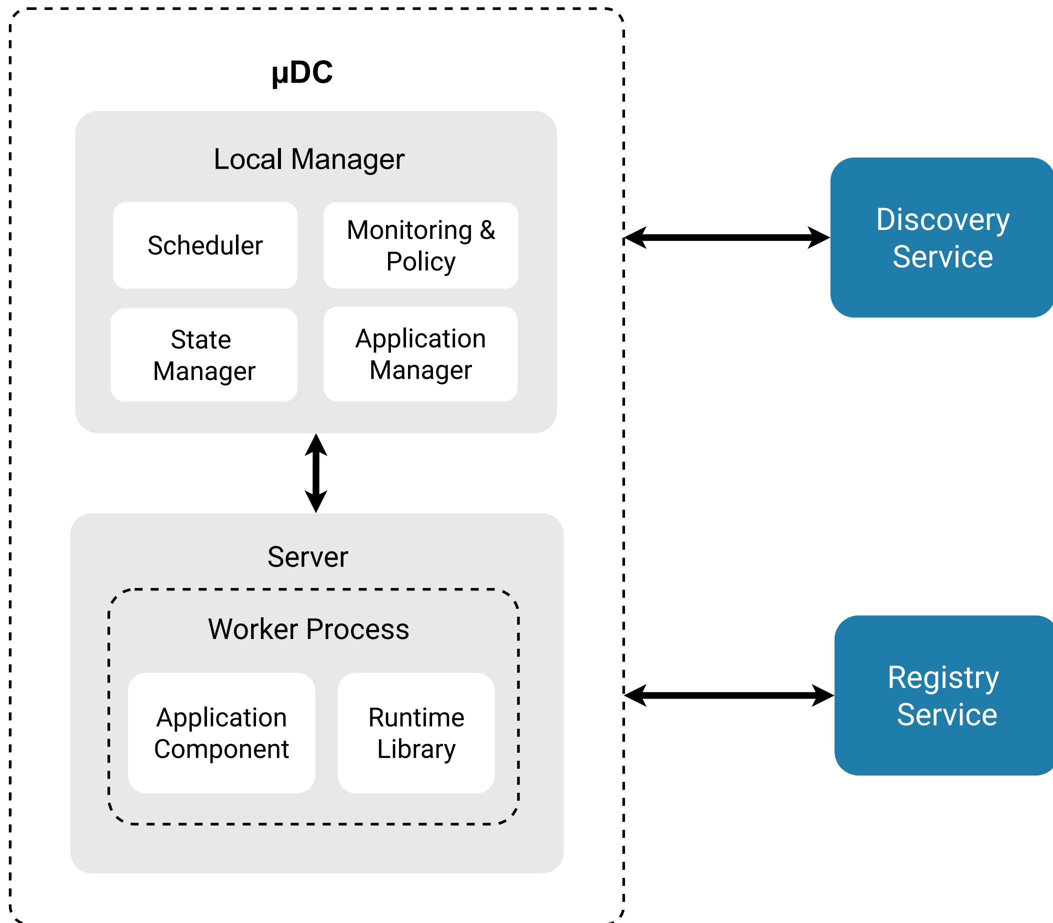

Figure 6.1: Foglets architecture. Foglets comprises four components: the registry service, the discovery service, the *local manager*, and the worker process. The registry service and discovery service have geo-distributed instances. There is one *local manager* per µDC. Each server can run multiple worker processes, one per application component of potentially different applications.

provide hints of the identifier for their network access point, such that better responses can be given, as network location is a better approximation than a geographical coordinate.

The registry service contains the binaries and configurations for the applications that will be launched on the *Foglets* infrastructure. There are replicas of the registry service in different geographical areas. As mentioned in a previous section (section 3.7), each application has a unique *appkey*. The registry service maintains the binary images of all the application components in the DFG, with the images indexed by both the application and a node identifier for the DFG. The discovery and registry services may be collocated on the same physical server.

Additionally, the *worker process* carries out the functionality contained in a particular application component assigned to it. The worker process combines the runtime library (explained in section 4.5) and the application component executable. In *Foglets*, the *worker process* is run inside a container, and there is an instance for each application component instance running in the system.

6.2 Workflow

To better understand the execution of applications in a decentralized context, let us discuss the simple missing child application presented in section 2.3. The missing child recognition has four main stages: video filtering, person detection, face recognition, and alerts. The workflow for developing and installing the application using Foglets is a 2-step process as follows:

1. The developer writes the application logic for each of the stages: filter, detector, face recognizer, and alert, as well as the handlers for each of the four levels.

2. The developer creates the binary container images for each of the application components (filter, detector, face, and alert), and then they use a system-wide unique *appkey* to register the container images (with the associated configurations) with the registry service, and optionally define a set of geographical locations.

The *Foglets* runtime will ensure that the application images and configurations are available in the corresponding registries for the hinted geographical locations (if given). Note that neither the resources nor the worker processes for the application components are provisioned at the installation time. Instead, such provisioning will occur incrementally based on the application dynamism, which is the crux of the dynamic discovery and deployment protocol to be discussed in the following sections.

Once a client appears, it will send a tuple containing both the *appkey* and its location, which trigger the actual deployment. *Foglets* will ensure that the computational resources are running in the required geographical region based on the application's latency requirements (if there is a µDC with enough capacity in the required areas).

6.3 Local deployments and peer-to-peer coordination

In this section, we present the two decentralized mechanisms used by Foglets to discover and deploy application components, as well as to join already running applications components.

6.3.1 Discovery and deployment protocol

Discovery involves finding the µDC/DC (matching the application constraints) at the right geographical/network location for hosting an application component. On the other hand, deployment has to do with starting a *Docker container* in a µDC/DC to run the *Worker process*. The *worker process* will carry out the work of the application component.

The *local manager* maintains the status of each application for the given µDC/DC. The status is modeled as a state machine, with the potential states being: *READY* - *DEPLOYED* (RD), *READY* (R), and *BUSY* (B). Both R and RD indicate that the µDC/DC can host the application component. Additionally, RD indicates that the required application component is already running at this location (i.e., a container with that application component is already present there), which can be advantageous in reducing the latency for the container provisioning. On the other hand, the B state marks that the µDC/DC does not have the resource capacity to accommodate any new deployment requests.

Resource discovery and provisioning occur incrementally in Foglets. For example, the first time an application component or a client attempts a *send_up*, the runtime library contacts the discovery service to obtain a list of µDCs/DCs in proximity that can provide the required E2E latency and resource constraints for the next downstream node of the DFG. Then, the runtime library executes a 2-phase join protocol to choose the next location to run the downstream application component from the list (Figure 6.4). The pseudo-code for the discovery and deployment algorithm is shown in Figure 6.2.

Upon getting the list of possible candidate µDCs/DCs, the runtime library sends a ping message to each of them. Each candidate responds with their node-id (n) and state (s). Depending on the response, there are three possible scenarios:

1: $candidates \leftarrow$ from Discovery Service
2: Send Discovery ping to each candidate
3: The candidate responds back its Id n and state s,
4: $s \in$ {READY-DEPLOYED,READY,BUSY}
5: Let R be the set of responses (n, s) $\forall$ candidates with n the Id and s the state.
6: Let $S = \{s|s = state(r), \forall r \in R\}$, be the set of all the received states.
7: **if** READY-DEPLOYED $\in$ S **then**
8: run the Join Protocol in Figure 6.3
9: **return**
10: **else** ▷ Application is not deployed in area
11: **if** READY $\in S$ **then**
12: $R_{READY} \leftarrow \{c|c \in candidates, state(c) =$ READY$\}$
13: Obtain the best candidate $c_{closest}$ from R_{READY}
14: Send a DEPLOY container message to $c_{closest}$
15: **else** ▷ No available resources in area
16: Restart the Discovery and Deployment algorithm, increasing the geographical area to be queried from the Discovery Service.
17: **end if**
18: **end if**
19: $r \leftarrow$ response from join to bc.
20: **if** $r = Accept$ **then**
21: Select bc as the next location to host the application component
22: **return**
23: **else**
24: $R_{READY} \leftarrow R_{READY} \setminus \{c_{closest}\}$
25: **if** R_{READY} is not empty **then**
26: **go to** Figure 13
27: **else** ▷ No location with available resources
28: Restart the Discovery and Deployment algorithm, increasing the geographical area to be queried from the Discovery Service.
29: **end if**
30: **end if**

Figure 6.2: Discovery and Deployment Protocol

1. The set of potential µDCs/DCs that have the application component already deployed (s = READY-DEPLOYED) in a container is non-empty (Line 8 of Figure 6.2). So, in this case, the Join protocol is directly called (Figure 6.3).

2. There are no µDCs/DCs with the application component already running. However, some locations are ready (s = READY) to accept a deployment (Line 12 of Figure 6.2). For this scenario, the runtime library chooses the best candidate location from the set of READY locations to deploy the application component as follows:

$$R_{READY} = \{c|c \in R, response(c) = READY\}. \tag{6.1}$$

   Then, choose the closest location (either based on the current location or with a projection to the future), $c_{closest}$, using Equation (6.2),

$$\min_{\forall e \in R_{READY}} distance(e, client), \tag{6.2}$$

   and initiate the second phase of the Deployment protocol (Line 14 of Figure 6.2), wherein the runtime library sends a DEPLOY message to $c_{closest}$ and waits for the response, which is either an $ACCEPT$ or $REJECT$. If the response is $ACCEPT$, the runtime library has successfully joined the downstream application component. If the response is $REJECT$, then the runtime library chooses the next closest candidate location in the set $R_{READY} \setminus \{c_{closest}\}$ and sends a DEPLOY message to it. If all the candidates send $REJECT$ responses, the Discovery algorithm is reinitialized with a bigger geographical area (currently, we double the radius on each iteration).

3. All the candidate µDC are fully committed (s = BUSY) and cannot accept any more requests. In this case, the Discovery algorithm is restarted with a bigger geographical search area.

6.3.2 Join protocol

The join protocol is depicted in Figure 6.4, and Figure 6.3 gives the corresponding pseudocode. First, the runtime library chooses the best candidate bc, the geographically closest, to send a join message as described in Equation (6.3), with the assumption that closer datacenters are more likely to provide the required latency.

$$\min_{\forall e \in W_{READY-D}} distance(e, client) \tag{6.3}$$

1: **function** JOIN PROTOCOL($W_{READY-D}$)
2: Obtain best candidate bc from the set $W_{READY-D} = \{c|c \in R, state(c) = READY - DEPLOYED\}$
3: Send a Join message to bc
4: Wait for response w
5: **if** w is $ACCEPT$ **then**
6: select bc as the parent and start connection
7: **else**
8: $W_{READY-D} \leftarrow W_{READY-D} \setminus \{bc\}$
9: **if** $|W_{READY-D}| > 0$ **then** ▷ Set is not empty
10: Choose the next best candidate bc in $W_{READY-D}$
11: **go to** Figure 3
12: **else**
13: Restart the Discovery and Deployment Algorithm
14: **end if**
15: **end if**
16: **end function**

Figure 6.3: Join Protocol

If there are many candidates with equivalent distances, their current load conditions could be considered in the choice. When the candidate *local manager* receives the join request for an application component already running, it queries the worker process associated with that application component. If the Worker process is ready to accept this new connection, an $ACCEPT$ response is sent to the requesting application component. On the other hand, if the load conditions have changed since the first phase of the Discovery protocol, the Worker process may decline the join request. Then, the candidate node would reply with a $REJECT$ response to the application component. In this scenario, the runtime library will try the next best candidate (Lines 8-11 of Figure 6.3). Additionally, it is conceivable that the network state may have changed during the execution of the join protocol, in which case the Discovery and Deployment protocol is started all over again (Line 13 of Figure 6.3) if the result is not satisfactory concerning the application constraints, such that even if Equation (6.3) is not a good proxy, it would eventually find a suitable µDC candidate.

The design of this incremental application deployment algorithm ensures highly adaptive and elastic resource utilization driven by application dynamics and QoS needs. Application components are deployed at a µDC/DC, only if an upstream application component executes a *send_up* message. Using this dynamic resource discovery protocol, the decentralized design incrementally maps the DFG of an application onto the physically geo-

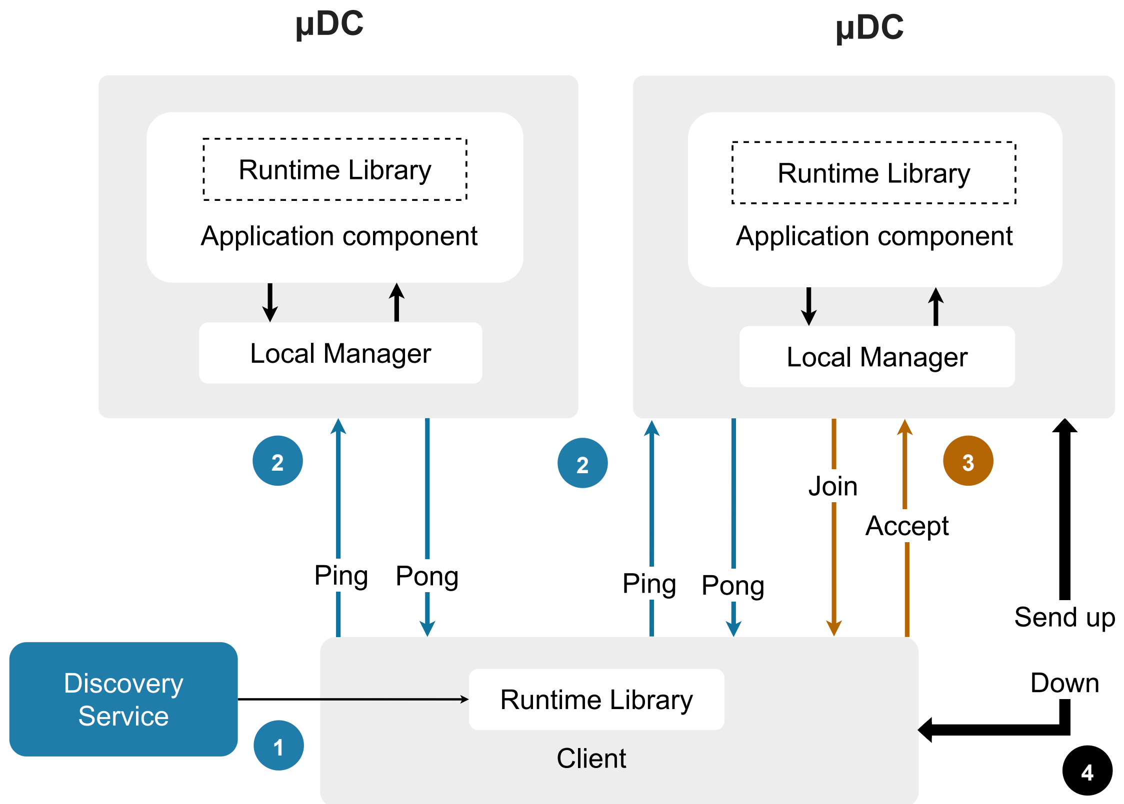

Figure 6.4: Join Protocol: the Discovery service gives a list of µDCs to the requesting client node. The protocol pictorially shown above results in the client choosing a parent to join from the list.

distributed computational infrastructure. Then, the *local manager* in a µDC/DC deploys an application component by launching a Docker container with the *Worker process* to carry out the functionality for that node in the DFG.

6.4 Migrations

In Foglets, migration of an application component from one µDC to another may be needed due to two main reasons:

Meeting QoS expectations: many situation-awareness applications involve mobile clients (*e.g.*, autonomous cars). In those scenarios, the µDCs hosting the application components for processing the client cannot be statically predefined. Instead, the selection of a µDC should be performed dynamically to adjust to the client's mobility patterns. Thus, mobility patterns that modify the E2E processing latency (requirement **R4**) could trigger the migration of application components.

Load balancing: in Foglets, a given µDC could be hosting application components from multiple situation awareness applications. Depending on the situation (e.g., traffic incident), a particular µDC could experience resource pressure. Thus, resource pressure at a µDC could trigger the migration of application components.

Additionally, there are two inter-related aspects concerning migration in Foglets:

Computation Migration: it is related to changing the µDC handling a client or an application component, either due to QoS or load-balancing considerations. To facilitate computation migration, Foglets expects the application to provide the two migration handlers described in section 3.8.2, which are executed when migration starts at the current application component instance, and when the migration is completed in the application component instance in the new µDC. The pre-migration handler saves any remaining soft state, and the post-migration handler initializes the application component instance for the corresponding client(s). Once the computation transfer is complete, switching the downstream node (or client) to the new application component instance is safe. The new application component instance will start processing *send_up* calls from this transferred upstream node when the initialization is complete.

State Migration: this relates to the persistent data generated by an application component (in the object store mentioned in section 3.5.3), which must be made available in case the client is migrated to a new µDC/DC for that node in the DFG.

The state migration is done in parallel with execution in the new node.

6.4.1 QoS-driven migration

Mobility is one of the main drivers in the development of new technologies. A good example of this trend is autonomous cars and the massive amount of computations required. That is why autonomous cars can benefit from situation awareness applications to perform more intelligent routing and improve the quality of decisions (*e.g.*, detection accuracy).

Mobility carries additional complexities, in which clients could drift away from the hosting µDC's location, which can adversely impact the communication latency. Furthermore, given that the µDCs are geo-distributed, it is also likely that clients become closer to a different µDC, requiring the system mechanisms for migrating the state and the computation to the new closer µDC. Equivalent to section 3.4.1, in Foglets, QoS is specified as an upper bound T on the E2E latency from the client to a given application component

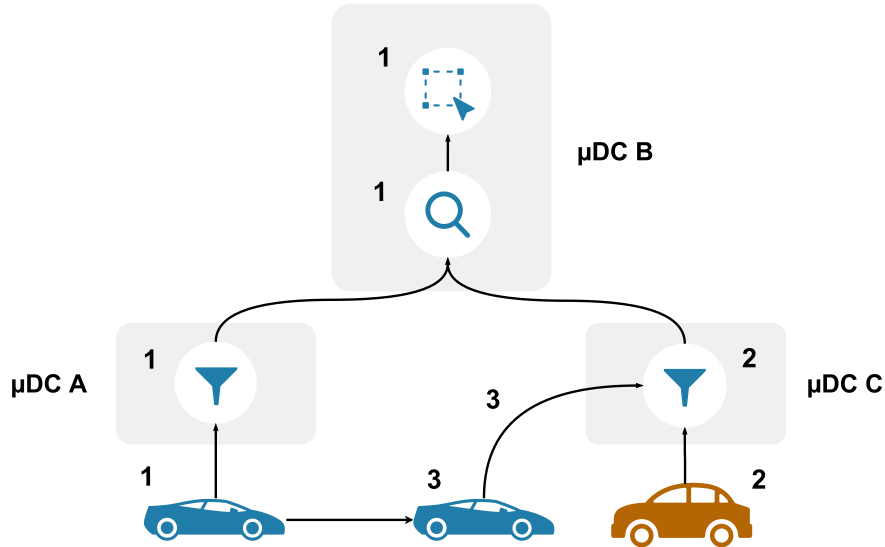

Figure 6.5: Quality-of-Service migration. Foglets migration moves latency-sensitive components to a more suitable micro-datacenter if the latency exceeds the threshold (*i.e.*, $\alpha \cdot T$ for proactive migration). Initially, in step 1, the blue car appears, and it is deployed across micro-datacenter *A* and *B*. At around the same time, in step 2, the brown car appears and gets assigned to micro-datacenter *C* and *B*, sharing some of the components. Finally, in step 3, the blue car moves away, causing the latency to increase. Consequently, Foglets proactively migrates the first component in micro-datacenter *A* to the already deployed one in *B*.

in the DFG (*i.e.*, the latency staleness $S(i, j)$ in chapter 3). We discuss two mechanisms implemented in Foglets that allow QoS-driven migration: *proactive* and *reactive*.

Proactive migration

Proactive migration is a mechanism that tries to avoid E2E latency violations before they happen. There are two parameters associated with proactive migrations in the Foglets system: α and β (both between 0 and 1), with $\alpha < \beta$. The α and β values are chosen commensurate with the QoS needs. The runtime system uses default values for these parameters if an application does not explicitly specify them. Additionally, the Worker process in Foglets has a one-to-one relationship with an application component in the DFG. Therefore, it is aware of the QoS requirements (E2E latency bound T) of the application component bound to it.

The Worker process continually monitors the actual E2E latency experienced on a *send-up* and from health-check ping messages from its upstream nodes (and aggregated from all

the downstream nodes). When the E2E latency from a given upstream node goes above a threshold, $\alpha \cdot T$, proactive migration starts. The Worker process at the current application component will find a suitable candidate substitute for an instance of its type from the list of available neighbors obtained from the Discovery service. An example and its description of migration can be seen in Figure 6.5. The choice is facilitated by the fact that each Worker periodically exchanges ping times (*i.e.*, round-trip network latency) and resource utilization information with its neighbors running the same node of the DFG.

First, only the *object store state* associated with this client will be migrated gradually in anticipation of the computation migration. We call this *state replication*. If and when the actual E2E latency goes above the second threshold, $\beta{\cdot}T$, then a decision is made to migrate the computation itself. In either case, the choice of the µDC to migrate to is based on several factors, including the geo-location of the µDC hosting the upstream node instance relative to the future µDC, the measured ping latencies, and the capacity constraints of the future µDC (measured by available uncommitted resources such as CPU and memory, which can be obtained from statistics maintained by the *local manager* of the current µDC).

Upon a decision to migrate the client to a new application component, the Worker process sends a *Start Migration* message to the candidate's new application component instance. Upon receiving an $ACCEPT$ message from this future application component instance, migration will proceed in parallel to move the computation and the object store state. On the other hand, if a $REJECT$ message is received for the migration request, the Worker process will initiate *Start Migration* with the next best candidate in the list of potential µDCs/DCs. Once a new application component instance has been identified to migrate, the migration proceeds, as we described before, with the invocation of the application-specific handlers on the old and new instances to transfer the upstream client node to the new application component instance.

In parallel with moving the computation, Foglets also initiates moving the object store state of the client from the old to the new application component instance. Some of the state may have already been replicated proactively before the instances were switched. Now that the new application component instance is ready, the state can be *moved* fully to the new instance instead of being replicated.

There are several opportunities for optimization. First, not all state needs to be moved to the new instance. Since situation awareness applications tend to work mostly with recent data, moving the most recent historical data to the new application component instance may be sufficient. If needed, the new instance can demand-load older historical data as

described in section 6.4.2.

Reactive migration

It is possible that due to the overall system dynamics, the proactive migration described above does not happen promptly enough to adhere to the E2E latency requirements of the application. It could even be that an upstream application component has become unresponsive due to overload. In this case, reactive migration may be triggered by the downstream application component. The upstream application component instance will decide to find a new downstream instance by going through the Discovery protocol (Figure 6.2). Once a new upstream instance has been found, transferring the computation and object store state from the old instance to the new one will proceed exactly as in proactive migration. The only difference is that the new instance will contact the old instance to initiate the migration.

Due to the dynamic loads and mobility patterns of geo-distributed situation awareness applications, *Foglets* may not react fast enough to these changes to maintain the latency requirements for all the clients. If this situation arises and the latency increases over an acceptable threshold, or if there is intermittent coverage and a client appears in a completely new location, the client would restart the discovery and deploy control. Once an accept message is received, it would send a message to the *old downstream instance* indicating the endpoint for contacting the previous downstream node instance, and the same state replication algorithm is applied to migrate the state.

6.4.2 Application state management

The state replication is performed in data chunks of size M, from the most recent to the oldest. This heuristic is chosen under the assumption that situation-awareness applications are real-time and are more likely to use recent data. In addition, the size of the chunks M should be optimized to the characteristics of the network to improve efficiency. After each chunk is migrated, the new application component sends a message requesting the next chunk; the message also includes the oldest received element. This design allows synchronizing the two application components without a state machine for the migration process on the previous application component.

Not all the data chunks have to be available in the local storage of the current application component w_1. For example, when a client moves fast and frequently across µDCs, it could result in the object-store state being fragmented in multiple previous application component

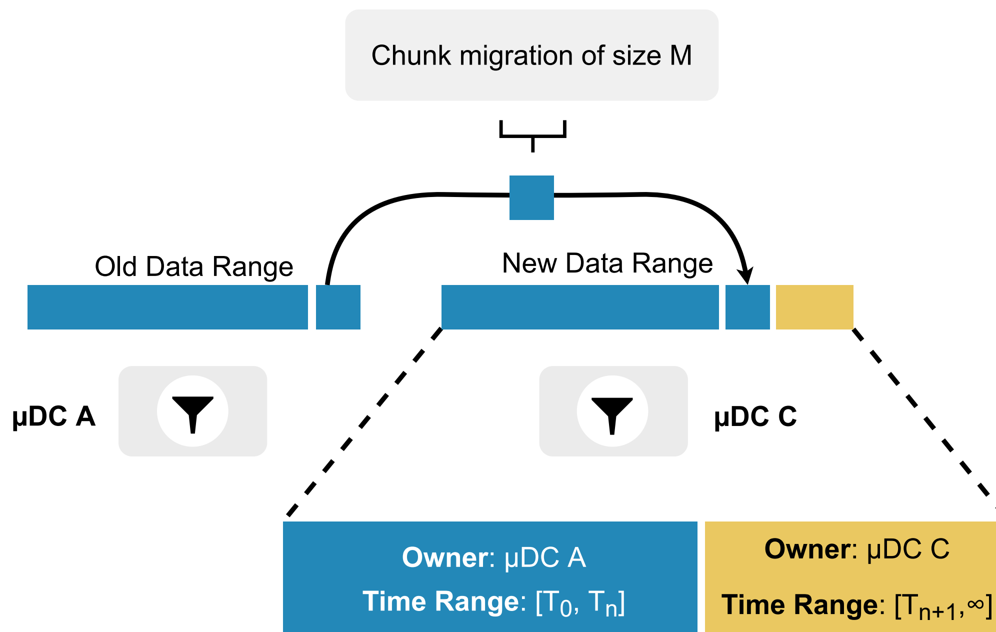

Figure 6.6: State Migration. The migration happens from the newest data towards the oldest data. Each step migrates a *chunk* of size *M*. Once the data is migrated, the data range is updated to show that the new instance in μDC C is the current owner of the data. If data for the old range $[T_0,T_n]$ is requested, the data would need to be fetched from the old instance in μDC B.

instances. Foglets does all the book-keeping to ensure that it has the trail of previous instances to retrieve historical items in any object-store in the presence of such migration. To keep track of the known data, we implement a data structure, the *data ranges*.

The *data ranges* structure contains information about non-overlapping ranges. Each range represents information generated at a certain time period and holds a pointer to the endpoint (*i.e*., the application component URI) where the data is currently stored (*i.e*., last stored). The *data ranges* are sorted decreasingly by time. By partitioning the data into ranges and holding a pointer to its current owner, Foglets can migrate an application component without migrating the whole application state to the new application component instance before performing the handoff to the new instance. Foglets perform both operations computation and state migration concurrently, which reduces the downtime of the handoff.

Given that the data of an application is not always locally available, the runtime library then needs to perform two different types of operations:

1. When an application component requests a specific key (and time), the runtime library will provide the value if it is already locally available. Otherwise, if the data is not yet available, it blocks the function call from the application component, it does an on-demand migration of that chunk of data (similar to a page fault in post-copy migration [65]), and it finally returns the requested data.

2. In the background, the old instance gradually migrates its chunks to the new instance.

Once a migration of a given chunk is completed, the *data range* is updated such that the owner is the current application component. The *data range* data structure needs to be synchronously migrated as part of the computation migration before a handoff can occur. Periodically during the state replication, the *data range* data structure is compacted such that consecutive data ranges that belong to the same owner are merged into one range. Furthermore, each application component keeps the metadata of data ranges that were previously proactively migrated. If a complete migration happens in the future, it may need to request for small gaps created between consecutive proactive state replications and merge them into the data structure. Once a full migration is initiated, we should stop all the proactive state replications until the new application component instance is fully updated, and then the corresponding neighbors of the new instance can be updated with the missing data ranges that are not yet owned by the current instance, from newer to older. Older owners of data can garbage collect the remaining state after a time threshold to reduce memory pressure, as this is expected to be soft state that is only meaningful for a short time.

6.4.3 Peer-to-peer coordination

The system requires storing information about other possible µDC as potential candidates for future migrations to correctly implement the first two types of latency-driven procedures (*i.e.*, state migration and proactive migration). *Foglets* leverages the geo-distributed characteristics of the network to implement the dissemination of information, which shows that with high probability, a given client would only move to a µDC that is a physical neighbor of the current hosting µDC. The only condition on which this is violated is if the client is turned off or has intermittent activity and jumps to a completely different area afterward. However, this condition is not an issue given that it would fall into the third type of migration (*i.e.*, reactive migration), and any of the prediction algorithms would not behave correctly under these circumstances, with the final objective of the first two types of migration being invalid.

In order to improve the knowledge of the neighbors, each runtime library in an application component will periodically, with a period t_d, send a message to the Discovery service to obtain all the other µDCs/DCs that are physically close to it. Then using this list, the runtime library would ping their µDC's *local manager* neighbors with a period t_n, where $t_n < t_d$. The ping contains information about the current average processing time and average latency in the application component. The *local manager* will then forward the message to the local instances of that application component (*i.e.*, Worker processes running the same node of the DFG). This forwarding is required, given that is no central database containing all the deployed application components, only information about µDCs. The neighbors' information is used, together with the locations of the hosting µDCs, and the current clients' location and velocity, to find a suitable new location to migrate it if required.

6.5 Dynamic resource reallocation: workload-driven migration

Multiple applications may be collocated in the same µDC/DC. Bursty resource requirements of one application could detrimentally affect the performance of other applications in meeting their respective QoS constraints. Foglets provides a mechanism for workload-driven migration to ameliorate these pressures. The *local manager* in each µDC keeps statistics on resource usage by all the containers deployed at the location to support workload-driven migration. In addition, the runtime library associated with these containers periodically reports their respective resource usages (CPU, memory, missed deadlines—if

any) and geospatial information of the upstream clients they are in charge of. Foglets runtime uses the stats provided by the *local manager* to make migration decisions if it finds that a server is overloaded.

While the QoS-conscious migration techniques deal with individual (upstream) client migrations, workload-driven migration does entire container migration when possible. The system sorts the containers by the number of clients each container handles. It chooses the top γ containers from this sorted list as candidates for migration. Using the geographical location information of the upstream clients catered to by the containers, Foglets chooses to migrate containers with upstream clients farther away from the location of the hosting µDC. The intuition is that such upstream clients are drifting away and likely moving to a different instance in the short future. Like proactive migration, an overloaded µDC pings its neighbors in different geo-locations to find a home for the containers it wants to migrate. Migrating whole containers will help reduce the load on the depressed µDC.

The Foglets runtime allows collocating applications component instances in the same computational resources in a µDC/DC. If there is an increase in the popularity of a given application, it could have repercussions on the performance of the other residing applications. An example would be an application that performs real-time video processing from smart-glasses video feeds. During a social gathering, such as a concert, there could be a peak in the consumption of resources by other applications. There is a need for a mechanism for either obtaining new resources or reallocating some or all of the other collocated application instances, which Foglets provides as part of its workload-driven adaptation. For workload-driven adaptation, the *local manager* monitors the workload at each container (*i.e.*, application component) running in the local servers. Foglets triggers a migration procedure when the pressure on the local resources (CPU, memory, and storage) is too big and affects the application's performance.

To be able to accomplish workload-driven migration, the worker runtime (*i.e.*, the runtime library) needs to constantly send monitoring information to the *local manager* that deployed them, with up-to-date information such as:

- Total percentage of memory and CPU used.
- Total missed deadlines.
- The number of clients.
- Average location of the clients.

The current mechanism used by Foglets is to select the container with the least number of clients that have the farthest average distance. The system first sorts by the number of clients, select the top γ containers, and then selects the one farthest away. There are two main reasons for using such an algorithm. First, clients drifting farther away from the original geo-location of the application component are more likely to be migrated by latency-drive migration and less likely to move closer to the µDC. Second, selecting the container with the least amount of clients helps to improve locality. By grouping clients of the same application, Foglets can improve hardware utilization (i.e., because of fewer containers running the same application). If the µDC selected as a migration target does not have enough capacity to support all clients, the system tries to contact an alternate µDC by following a list of candidates sorted by Euclidean distance. Once a new µDC/DC is selected, the same algorithm for proactive migration (described in the previous section) is used to migrate each client.

The hardware requirements for the *local manager* in a µDC are small, given that it does not do any complex computation. Its main job is to forward messages and deploy and migrate containers. The algorithm for migrating the containers is a lot simpler than the one for managing the migration of clients.

6.6 Implementation

The Foglets system was implemented using C++ with the operating system Ubuntu 14.04. The communication layer is implemented using the ZMQ [66] framework for message transmission between components and the Protobuf library from Google [67] for data serialization. In addition, the Foglets implementation uses the Docker container runtime [68] and RocksDB [69]. The *local manager* manages the Docker runtime on each server in a µDC. The base image used to develop the container images with the Foglets library and the Worker process runtime is *Ubuntu* (version 14.04).

Instantiating and creating a new instance of a container is fast, given that the images created by a developer will layer their binaries on top of the base Foglets image. This design decision allows a developer to test the different characteristics of the system locally in their development system, without the need to deploy their application on top of the real hardware. Further, if the Foglets container images are pre-installed in a µDC local registry, then to start an application component on that node, only the delta (the specific application) must be pulled from the registry service, reducing the overhead.

RocksDB has two main features that help efficiently implement the object store: *prefix*

iterator, and *read-only access mode*. [1] The prefix iterator allows fast access to the object store (using the *time* of the write as the prefix in our implementation), and the read-only access mode eliminates the need for locks during migration of the object store state.

Finally, Foglets geographically replicates the Discovery service for scalability since it only maintains connection endpoints that are updated infrequently, namely the µDC/DC (and not the client devices themselves nor the application components).

6.7 Evaluations

In this section, we evaluate the performance and functionalities of Foglets. First, we measure the costs, in time, associated with launching a container in the Foglets runtime system. Then, using a workload generated with the SUMO traffic simulator [70] to drive our system, we conduct experiments to measure the time for the incremental deployment operations of Foglets (*i.e*., Discovery-Join and Discovery-Deployment) for launching application components in the geo-distributed infrastructure.

Next, we conduct experiments on the migration component of the Foglets system. The first experiment measures the basic cost of switching from one application component instance to another. We then conduct experiments to show the efficacy of the migration operations of Foglets under two conditions: (a) dynamic workload-driven reactive migration and (b) proactive migration when E2E latency constraints are not met.

6.7.1 Platform

Our experimental platform is a cluster of four Penguin Relion 1752 nodes interconnected by a 10 Gbps Ethernet switch. Each Penguin server contains a 2-socket, 6-core, 2.66GHz Intel X5650 hyper-threaded processor with a memory of 48GB RAM. These four hardware platforms are used to emulate 16 µDCs. We run Ubuntu 14.04 on all the nodes as the base OS, and the containers are running directly on top of this OS. We also emulate the discovery server and registry service on these machines. Finally, we use an auxiliary machine that serves as the workload generator for our emulated geo-distributed infrastructure. This auxiliary node sends messages to the µDCs emulating the client inputs (*i.e*., position information and video streams) from vehicles moving in the city of Atlanta.

[1] The prefix iterator is no longer supported in newer versions of RocksDB. Therefore, version 4.8 was used in these evaluations.

Table 6.1: Startup times for different configurations of Docker images

Docker Image	Time (s)
Debian	8.6
Ubuntu	1.07
Foglets base	1.1
Foglets application w/base	18.36
Foglets application already deployed	0.42

6.7.2 Starting the Foglets system and application components

The Foglets system implementation uses the Docker container runtime as one of its main building blocks. This section measures the times of "booting up" the Foglets runtime and a breakdown of each of its main components. The Foglets base Docker image (containing the runtime for the API calls, the communication libraries, and the Worker process to run the application) is built on top of the Ubuntu official docker image. The developers use the Foglets base image to implement their application components and as the image to be deployed at each location in the geo-distributed µDCs. We measured the times to start up each layer of this software stack needed at each µDC's server. As a baseline value, we use the time taken to download and execute two images from the official Docker container (*i.e.*, Ubuntu and Debian). We select these two images, given that the Ubuntu image is constructed on top of the Debian image as a delta, and we highlight the importance of leveraging the union filesystem [71] used by the Docker container runtime.

The numbers shown in Table Table 6.1 are the average of 100 runs. In the remaining of this subsection, we present the base costs of "booting up" the Foglets runtime:

1. Pulling the full Debian image from the repository and starting its execution takes 8.6 seconds on average. Then, with the Debian image already in the system, the docker runtime only needs to download the additional layers to form the Ubuntu image with an average cost to download and start of 1.07 seconds.

2. The Foglets system base image is developed on top of the Ubuntu image. It contains all the required libraries and the runtime system for implementing the Foglets primitives. If the Ubuntu image is already present in the host system, pulling the image and booting up the system takes, on average, 1.1 seconds.

3. The reference application we use in our evaluation is the vehicular traffic simulation on Atlanta streets. The application consists of vehicles sending their positional information and video to their respective µDC. The application uses OpenCV for

processing the video feeds. Consequently, the library needed for the application is large and serves as a good reference application for our experimental studies to show the efficacy of our system for dealing with large application images to be launched on the µDCs. The application with all the libraries and the Foglets handlers is 2.1 GB. Due to its size, downloading and starting the image, even with the Foglets base image present, takes 18.36 seconds. However, downloading the application image needs to happen only once when the application is started or during the bootstrap of the Foglets system.

4. If the application image is already locally available, launching it takes only 0.42 seconds. To bootup Foglets, we first distribute the binaries of *local managers* to all µDC. On average, starting the *local managers* in a µDC has a latency of 4.17 ms.

5. The clients do not need to use a container to run their application, as they can be run bare metal in the available OS in the client; starting the Worker process bare metal has a latency of 40 ms.

6. The Worker process binary size is 45 MB, and the *local manager* binary size is 43.4 MB. We measured a raw throughput of 50 MB/s between the nodes of our experimental platform. Thus, it would take approximately 1 second to send the binaries to their corresponding nodes. Our measured times for bootup are in agreement with the raw throughput measurement. It takes about 30 seconds for the *local manager* binary and the container image to be downloaded (which is a summation of all the numbers in Table 6.1), both of which can be done in parallel.

This section showed how fast we could deploy the runtime, add new µDCs, and deploy the application components containers.

6.7.3 Microbenchmarks

This subsection evaluates the cost of the main operations in the Foglet system by simulating the movement of vehicles as clients.

Workload

As we mentioned in the previous subsection, we use a vehicular simulation as the driver application for our experimental study. We simulate the movement of the cars using the SUMO traffic simulator [70], which can model realistic traffic patterns of vehicles on an

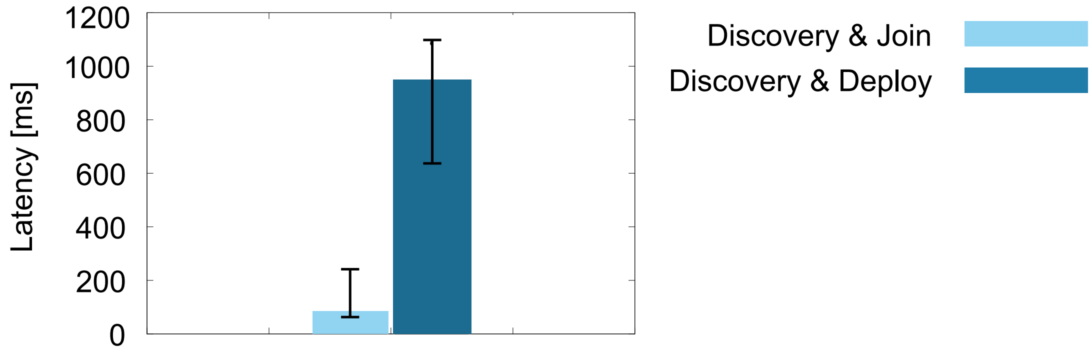

Figure 6.7: Comparison of Discovery-Join and Discovery-Deployment Operations. Error bars represent the 25% and 75% percentiles.

Atlanta OpenStreetMap graph [72]. Our simulation is a snapshot of the traffic in a rectangular grid of the Atlanta city (7.7 km x 5.7 km) for 10 minutes using the road network graph and 600 vehicles for each simulation run (on average). We observed approximately 110,000 events for each simulation run, meaning that each vehicle sends 184 events on average, including both locations and images to be processed. At any instance of time in the simulation, there are, on average, one hundred cars in the area covered by the simulation. Using a grid structure, we divide the parts of the city covered by the simulation into 16 geographic regions and assign a μDC to cover the client inputs for each region. As we noted earlier, we emulate the 16 μDCs on the 4 Penguin machines. The distance of the car to the μDC's location is included as part of the latency calculation to emulate the movement of the cars into the execution of the Foglets system, as shown in equation (Equation 6.4).

$$latency = measured\ latency + \epsilon \cdot distance, \tag{6.4}$$

where epsilon is chosen such that if a car moves out of the grid section that it is currently in, it is most likely violating the QoS requirement (expressed as an E2E latency constraint). Similarly, *Foglets* depends on having a function that roughly maps the network/geographical location of the clients with the expected latency to the μDCs to decide better when and to whom to migrate a component.

Discovery, deployment, and join operations

Figure 6.7 compares the cost of the protocols for creating and using an application by the client: discovery-join and discovery-deployment. We limit the number of candidates to four each time we try to either deploy or join the system. If unsuccessful, the runtime library tries the next best candidate in the list or increases the geographical area to be queried. As a reference, the total latency for performing a null RPC is around 4 ms (a send_up followed by a send_down, shown in Figure 6.8).

As shown in Figure 6.7, there is considerable variance in the discovery and deployment measurements, where the lower 25% of the deployment operation takes less than 636 ms, and the higher 25% of the deployment operation has a latency of more than 1097 ms. Similarly, the lowest observed delay is close to the minimum possible of 42 ms, the time required to start a container, as shown in Table 6.1. The huge variance is due to the state machine used for implementing the discovery protocol, in which it waits for responses from all the *local managers*. A faster response can be obtained from the agents involved if the system is not loaded. The long tail that could happen in the deployment algorithm is upper-bounded by the timeout mentioned in Figure 6.2.

The Discovery-Join protocol (wherein the application is already running in the container) is much faster. The median for this is 72 ms, with the 75% percentile at 240 ms, mainly depending on how fast the *local manager* can respond to the new request in the first phase of the algorithm. The measurements include situations in which a JOIN request is returned with a rejection.

Migration operations

Figure 6.8 shows the average cost in milliseconds for selecting and changing to a new downstream application component instance for a client in proactive migration. The measurement is taken when there is no application state to be packaged and sent to the new application component instance (i.e., the on_migration_start handler is a null handler), and the application is already deployed in the new µDC. As shown in Figure 6.8, this operation takes roughly three times compared to a round-trip message. The time required for reactive migration is the same as the discovery protocol measurements shown in Figure 6.7 since it is initiated by the client (or upstream application component).

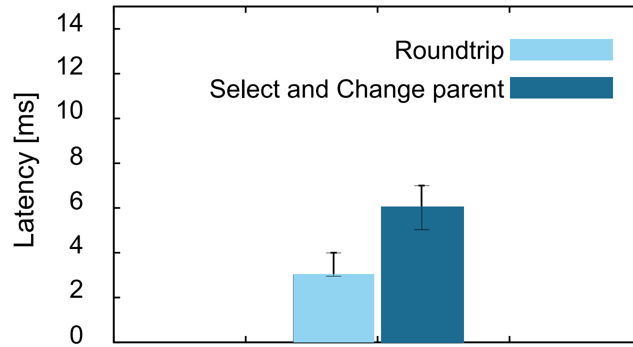

Figure 6.8: Proactive Migration Operation. As a point of comparison, we show the network round-trip time. Error bars represent the 25% and 75% percentiles.

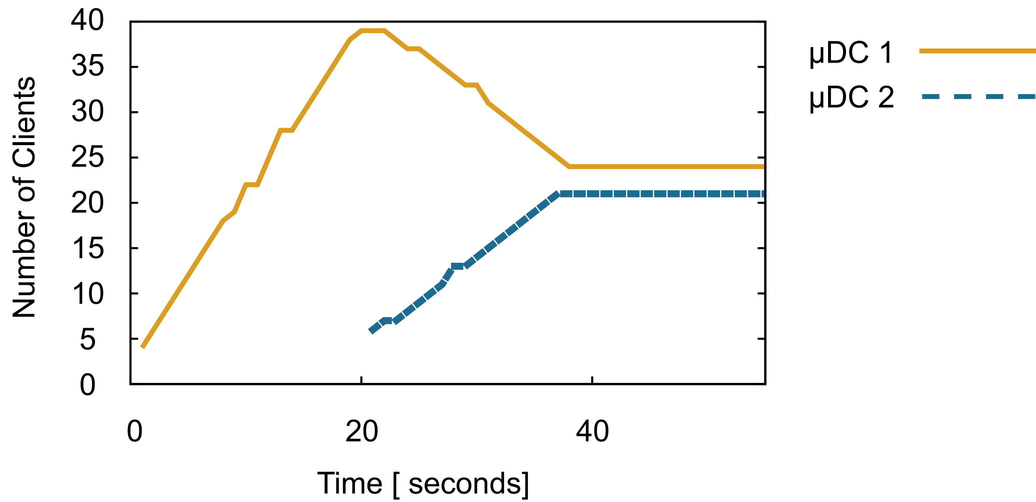

Figure 6.9: Workload Driven Migration. Over time μ*DC 2* accepts more clients to offload the work from μ*DC 1*.

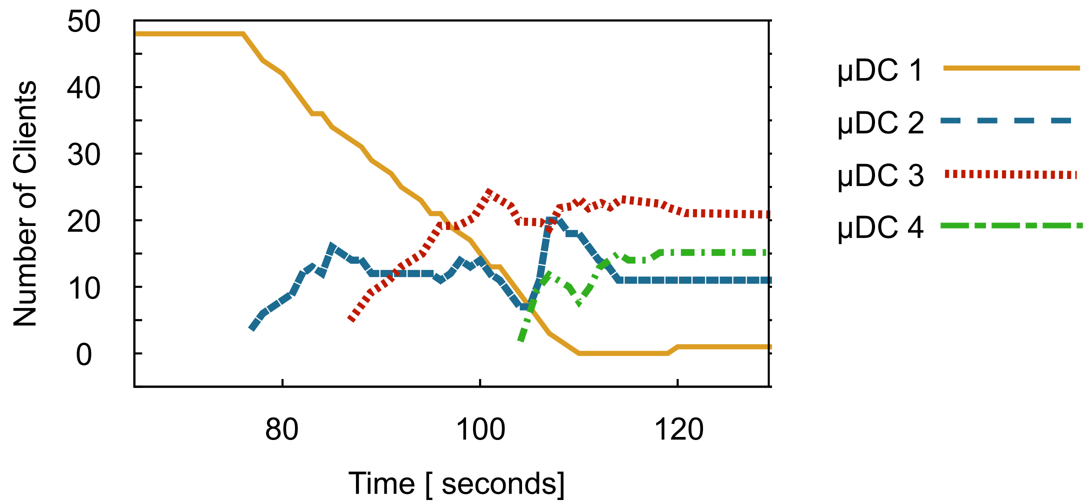

Figure 6.10: QoS-driven Proactive Migration. Over time the clients are migrated to the micro-datacenter that is geo-local to the clients moving in different directions.

6.7.4 Dynamic workload-driven migration

In this experiment, we demonstrate qualitatively and quantitatively the ability of Foglets to do workload-driven migration. The experiment uses two µDCs geographically close to each other. A client could, in principle, choose either of them as the location to host the application's DFG. We allow a new car to enter the geographical area every 0.5 seconds and stay put (i.e., it is stationary). The first car to appear would join one of the two µDCs (µDC *1* in Figure 6.9). Since *µDC 1* has a container already launched with the application, the subsequent cars entering the same geographical area will prefer to join the same µDC, as shown in Figure 6.9 at the 20 seconds mark. The capacity limit for *µDC 1* is reached when 39 cars have joined it. Subsequent join requests will be rejected, resulting in the next car joining *µDC 2*. At this point, we stop introducing new cars into the experiment. *µDC 1 and 2* exchange latency times and capacity information. Due to the resource pressure, *µDC 1* starts transferring some of its clients to *µDC 2*. Since the cars are not moving, the transfer is gradual, but in the steady-state, it can be seen that both µDC have roughly an equal number of clients. The migration is incremental to avoid oscillations between the µDCs and to have a stable migration control loop.

6.7.5 Proactive migration

In this experiment, we want to demonstrate qualitatively and quantitatively how Foglets does QoS-sensitive, proactive migration. The context is a traffic jam. Several vehicles are selecting the same µDC initially. However, as the traffic jam alleviates, the vehicles drive toward their respective destinations in different directions. We simulate this situation by first spawning some cars close to *µDC 1* and letting them stay put (*i.e.*, they are not moving). This behavior can be seen in the results shown in Figure 6.10 for the first several tens of seconds. Then, we make the cars move away in three different directions, starting around 70 seconds into the simulation. Due to the increasing distance from *µDC 1*, some of the clients start experiencing increasing latencies triggering proactive migration of the clients to the µDCs corresponding to the areas the cars are moving towards. As shown in Figure 6.10, Foglets can quickly migrate the application component instances, responding to an upsurge in the latency experienced by the clients. Similarly, it is observable that the number of vehicles stays constant in the graph, which is an indication that the application misses no messages and all the clients are successfully transferred from *µDC 1* to other close-by µDCs.

6.8 Chapter summary and limitations

This chapter proposed a control plane and runtime for the computational continuum extending from the sensors to the cloud called *Foglets*. It implemented the APIs presented in section 3.3 for application development and for placing the corresponding computation in the different µDCs/DCs commensurate with their latency properties. Foglets facilitates the primitives for communication between the application components and the algorithms for discovering and incremental deployment of resources commensurate with the application needs. It also provides mechanisms for QoS-sensitive and workload-sensitive migration of application components due to client mobility and application dynamism. Additionally, we introduced a complete implementation of Foglets that uses the Docker container runtime as the base substrate, and we presented performance results to showcase its effectiveness for situation awareness applications and geo-distributed computational resources.

The main limitation of Foglets is when operations require coordination across multiple µDCs. The two main operations that require coordination are load balancing and providing the spatial affinity requirement presented in section 3.4.3. For example, the instance uniqueness for each AoI cannot be efficiently provided in a fully decentralized

fashion without adding additional components or mechanisms. Furthermore, the peer-to-peer mechanisms used in this chapter do not guarantee strong consistency, which can cause the deployment of multiple instances for handling one AoI; the effects of decentralized management of AoI are compared in section 7.10.3. Similarly, load balancing in *Foglets* is performed in a greedy fashion, where each μDC tries to offload some application components when its local quota grows higher, with the restriction that migrations should not be performed frequently to avoid inefficient oscillations of application instances between μDCs given the stale information available in each μDC about other μDCs. A more optimal load balancing can be achieved with a global view of the available resources, which is important given the scarce available resources at μDCs. Therefore, chapter 7 presents additional components and mechanisms to efficiently support requirement **R3** (*i.e.*, spatial affinity) without losing support for the other requirements, as well as to maintain a global view to perform better long-term resource management.

CHAPTER 7
HYBRID ARCHITECTURE

In the previous chapter, we presented a fully decentralized control plane architecture. It focused on allowing autonomous decision-making for each µDC/DC (requirement **R1**). However, it only took a best-effort approach in providing the two semantic requirements of situation-awareness applications, namely spatial-affinity and E2E latency bounds (requirements **R3** and **R4**), given that it only had a myopic view of the available resources in the overall infrastructure.

Supporting both standalone and coordinated applications have conflicting needs that are tightly related to the latency and spatial affinity requirements, respectively. First, autonomous deployment necessitates decentralized state and autonomous decision-making based on locally available state. Second, coordinated applications need multi-µDC deployment and coordination that necessitate an orchestrating entity with a global knowledge of the system. Finally, an important additional problem is that µDC-specific performance monitoring alone is insufficient to provide E2E latency guarantees, as application DFG (Figure 3.3) may span multiple µDCs. This chapter addresses these conflicting requirements.

This chapter presents OneEdge, an agile control plane designed to meet all the requirements (section 4.2) of situation awareness applications and geo-distributed infrastructure. Specifically, it allows µDCs to take autonomous scheduling decisions without central coordination for standalone applications. Additionally, to cater to the needs of coordinated applications that rely on global knowledge for correct scheduling, OneEdge has a centralized component. The main focus of this chapter is the mechanisms required to provide hybrid control plane architecture for geo-distributed infrastructures that can efficiently combine autonomous decision-making at µDCs to minimize deployment latency for standalone applications, with centralized decision-making for scheduling coordinated applications. Most of the contents of this chapter were previously presented in OneEdge [22].

7.1 Insights and benefits of hybrid

The main contribution of OneEdge is a hybrid control plane with an efficient interaction between the logically global control plane manager and the independent per-µDC *local*

managers. The design tries to reach a balance between µDC autonomy and the use of global knowledge for optimization decisions when required. To achieve a good balance, we designed OneEdge's architecture based on three key principles:

Design principle 1: OneEdge exposes client location and application latency needs as *first-class citizens* to the control plane components. Each client's application deployment request contains the required information (*i.e.*, the client's geographical location). On the other hand, all the bookkeeping performed to the control plane's infrastructure state is geospatially organized. Further, SLA monitoring becomes an integral part of the control plane, taking placement and migration decisions based on the application's geographical context. This principle addresses the semantic limitations of current state-of-the-art architectures exposed in chapter 5.

Design principle 2: A two-level hybrid architecture can reconcile the opposing need for both decentralized and global decisions. This hybrid control plane architecture is composed of an autonomous manager per µDC and an overarching *global manager* performing complementary actions to meet the demands of both standalone and coordinated applications. Autonomous per-µDC *local managers* maintain the µDC's authoritative state and allow instant deployment of standalone applications without coordination with the *global manager*. On the other hand, the *global manager* maintains an *eventually-consistent* view of all the µDCs' states, which is used to improve semantically required adaptations (*i.e.*, latency and spatially driven migrations), cross-µDC application DFG deployments, and off-the-critical-path load balancing.

Design principle 3: Application deployment decisions should be primarily objective-driven rather than fully resource-driven. Managing and enforcing application E2E latency and spatial guarantees should be the control plane's duty. Thus, the control plane should not only schedule DFGs based on application objectives but also continuously maintain them via hierarchical monitoring. Per-application latency metrics must not only be collected locally at each µDC's *local manager* but also periodically processed at the *global manager* to assess SLO compliance. The control plane reassesses previous allocation decisions upon SLO violation detection and potentially migrates application components depending on the SLO violation's source (*e.g.*, resource scarcity or client mobility).

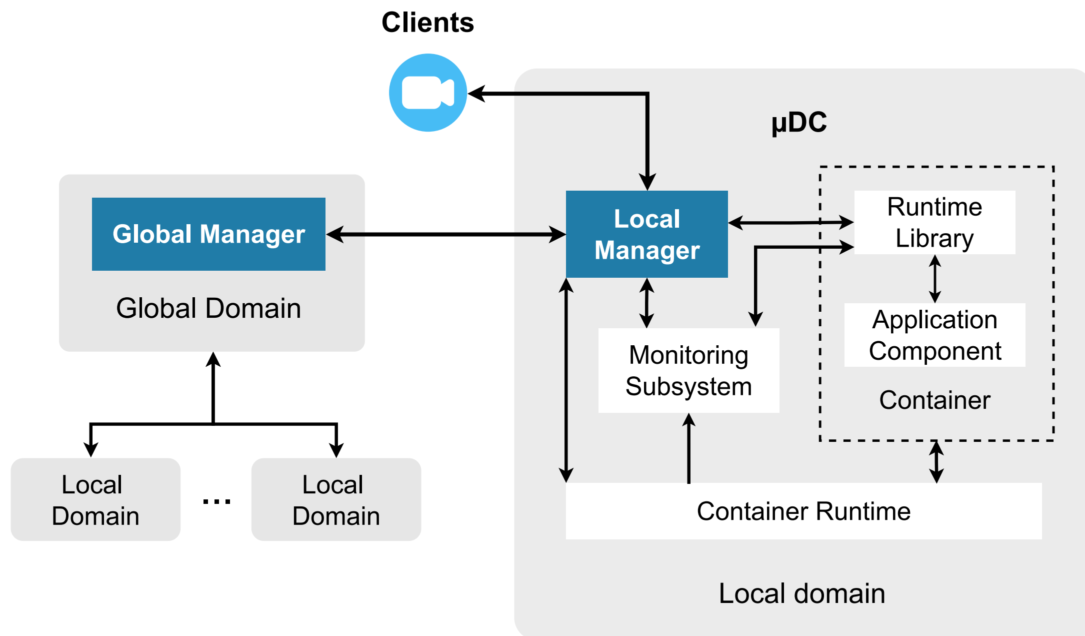

Figure 7.1: OneEdge's System Architecture. The global domain manager (left) coordinates with all the local domain managers in each µDC (right blow-up). Additionally, there are three more components in the in the local domain: monitoring subsystem, runtime library, and container runtime.

We now present the architecture of OneEdge, a hybrid control plane that leverages these design principles —and the mechanisms of previous architectures, centralized (chapter 5) and decentralized (chapter 6)— to meet the requirements outlined in section 4.2.

7.2 Architecture overview

Figure 7.1 highlights OneEdge's high-level architecture, composed of two top-level domains: *local* and *global*. The local domain is an autonomously managed instance of a geo-distributed infrastructure (*e.g.*, µDC, Cloud datacenter) that contains computational resources and cooperates with the global domain in control-plane decision making. On the other hand, the global domain is a logically centralized entity that makes system-wide deployment decisions for applications' DFGs spanning more than a single local domain or to adjust deployment decisions that were autonomously made at individual µDCs, for load balancing reasons.

The *global manager* monitors and orchestrates the aggregated infrastructure's state. Additionally, it performs actions both on (*proactive*) and off (*reactive*) an application de-

ployment's critical path. For example, for coordinated applications, the *global manager* selects each client matching to application instances and the placement of those application components to specific µDCs/DCs. Similarly, in the background, the *global manager* also monitors each µDC's resource usage and each application's measured performance compared to its SLO to drive resource reconfigurations and application migration decisions.

The system architecture of OneEdge consists of five components, shown in Figure 7.1, split into the global and local domains. In the global domain, there is only a logically global *manager* for the entire edge-cloud continuum infrastructure, which makes deployment decisions in coordination with the elements of the architecture within each µDC shown in Figure 7.1. In the local domain, within each µDC (which could be a micro-datacenter in a central office or a cloud datacenter), there are four elements:

***Local manager*:** The resource manager for a given µDC that acts as the coordination point between the *global manager* and the individual µDC.

Monitoring subsystem: The key driver of the responsiveness of the overall architecture. It continually collects metrics about the resource usage at the µDC and the health of each application component hosted in this location (*e.g.*, SLA violations).

Runtime library: Each µDC can host multiple applications (*i.e.*, it is multi-tenant), and there is a runtime library associated with each application component. It also collects statistics on the associated application component (*e.g.*, resource usage, SLA violations) and passes it on to the monitoring subsystem.

Container runtime: The container runtime is generic and uses an open-source platform (Docker [68]) for spawning and managing containers in each µDC/DC.

For responsive, autonomous control plane decisions without central coordination, the authoritative state is kept locally at each µDC. In addition, the *global manager* keeps an *eventually consistent* [73] aggregate state of the overall infrastructure to make scheduling decisions for multi-µDCs coordinated applications. The control plane takes decisions optimistically, given the eventually consistent construction of the aggregate state. Then, the decisions have to be ratified by the affected µDC's *local manager*. The mechanism for the ratification and coordination is presented in section 7.6. Finally, the application dynamic state management is handled in the same way as section 6.4.2.

The design partition aligns directly with the components presented previously in section 4.5. The components in section 4.5 were designed to provide the functionalities re-

quired without considering either the physical constraints or how they could be implemented in a performant manner. The following sections present efficient mechanisms for the interaction between the components presented in this section, as well as to cater to the requirements (section 4.2) of a geo-distributed infrastructure and situation-awareness applications.

7.3 Workflow

The client's requests to access an application on the geo-distributed infrastructure are always directed to the client's geographically proximal µDC by leveraging standard discovery services [74]. The developer labels each application as either "standalone" or "coordinated". The µDC's *local manager* handles deployment requests for standalone applications locally, avoiding WAN traversals and global coordination unless resource constraints prevent the local deployment. On the hand, deployment requests for coordinated applications are forwarded to the *global manager*. Figure 7.2 shows the *global manager*'s workflow for handling deployment requests received from *local managers*. The monitoring subsystem can also push reconfiguration requests into the request queue to avoid potential or current SLO violations detected through the monitoring statistics.

7.3.1 Local-domain overview

The right section of Figure 7.1 shows the components of the local domain. The *local manager* runs in one of the servers in the µDC. It receives deployment requests from the clients within the µDC's geographical range and from the global domain. The container runtime deploys each application component, each of which is deployed with a collocated runtime library. Each component's runtime library handles inter-µDC communication between application DFG stages spanning multiple µDCs. Finally, the monitoring subsystem continuously gathers metrics about the µDC's resource usage and metrics related to the SLOs of the application components hosted on that µDC. In this respect, it is similar to the monitoring component in chapter 6. The monitoring subsystem is further discussed in section 7.5.1.

7.3.2 Global-domain overview

OneEdge's *global manager* (Figure 7.2) plays two crucial roles. First, it determines the placement of cross-µDC application DFGs, a need typically associated with coordinated

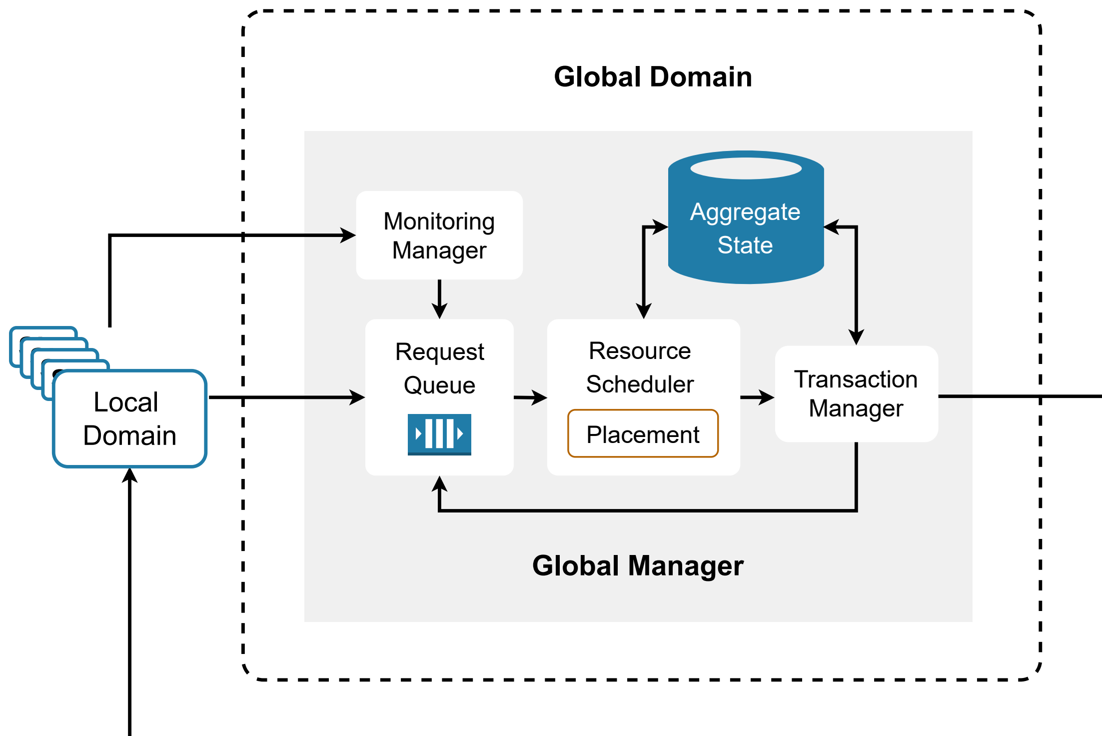

Figure 7.2: The global manager comprises five components: monitoring manager, request queue, resource scheduler, aggregate state, and transaction manager. Requests are added to the request queue by the local domains, monitoring, and transaction managers. The resource scheduler then processes the requests, and the transaction manager finally executes them.

applications. Second, it continuously monitors the entire infrastructure for significant load imbalances and E2E application SLO violations. When such incidents are detected, the controller will make reallocation and potentially migration decisions to ameliorate the problem. Different components of the architecture fulfill the two roles in Figure 7.2. The scheduler (section 7.4.2) and the transaction manager (section 7.6) handle the first role jointly with the two mechanisms presented in sections 7.4.1 and 7.6. The second role is mostly handled by the monitoring component (section 7.5.1) and the dynamic resource allocation mechanism (section 7.5.2).

7.4 Multi micro-datacenter mechanisms

Spatial affinity involves the need to coordinate across µDCs, given the need to share specific instances of pipelines across multiple clients. Additionally, given the limited set of resources available in each µDC, better load balancing mechanisms are needed to use the resources more efficiently. This section presents a mechanism to improve the load balance across µDCs by leveraging the global knowledge available in the global domain and the scheduling component that allows deploying application components across multiple µDCs.

7.4.1 Deflection

A *local manager* generally forwards new deployment requests for coordinated applications to the *global manager* and autonomously handles the local deployment of standalone application requests. However, when a µDC has high utilization of resources, even standalone application requests cannot be locally served and must be *deflected* to the *global manager*. We extend this deflection to create a new mechanism that improves the overall load balance of the geo-distributed infrastructure.

The fundamental intuition is to probabilistically upgrade requests to be processed like a coordinated application to improve utilization. The autonomous deployment of standalone applications improves deployment speed. However, it may lead to utilization skew and overloading of individual µDCs (*e.g.*, spectators with AR/VR devices at a ballgame connecting to the same proximal µDC). To address such skew, we introduce logical federations of µDCs by geographical area, called *cells*, and allow standalone requests to be deployed on any of the µDCs within the same cell. However, periodic load balancing by the *global manager* via redistribution among cell members is not enough for handling such workload skews. For example, suppose the scheduling decisions are performed per message,

which is possible for many standalone applications. In that case, long-term load balancing cannot perform efficient job-stealing between different µDCs, given that each operation is potentially short [75]. So, we allow the local domain to take proactive measurements to reduce the likelihood of overload and have more headroom for future actions by deflecting standalone requests before the resources are saturated.

To improve the load balance of regions with enough density of µDCs with short network proximity to each other, we propose a mechanism that deflects requests that would usually be served locally to be handled by the *global manager*. The µDCs of a cell use a *deflection* mechanism to leverage this flexibility and alleviate the effects of load skew. The *global manager* knows cell memberships. Therefore, when deflection conditions at a given µDC are met, the *local manager* forwards new standalone requests to the *global manager* instead of deploying them locally. In turn, the *global manager* determines which of the corresponding cell's µDCs to deploy the request that can meet the application SLO and provides better load balance across the cell.

There are two parameters dictating a µDC's deflection policy: *threshold* and *percentage*. The *threshold* specifies a low watermark, defined as the percentage of local resource utilization (*i.e.*, how full is the µDC), after which deflections will start happening. The deflection *percentage* represents the ratio of requests after the *threshold* that are upgraded to be handled as coordinated applications. All requests are deflected when the resource utilization goes above a high watermark (by default being full capacity of a µDC). OneEdge's deflection is inspired by a similar dynamic load shedding technique previously proposed in rack-scale systems [76]. As shown in the evaluations (section 7.10.2), this is a simple policy that helps to better distribute the load by leveraging the already available global knowledge. In addition to deflection, we also perform periodic global load balancing for long-term coordinated applications.

7.4.2 Scheduling

In the global domain, the *resource scheduler* processes the requests sequentially, making placement decisions to fit each request's requirements. The input to the resource scheduler is composed of the following parameters:

Type: either standalone or coordinated.

E2E latency SLO: inferred from the application DFG to be launched; it was defined when the developer uploaded the application to OneEdge infrastructure.

Spatial affinity: it is also inferred from the request-initiating client's GPS location and the application DFG. The GPS location is transformed into a specific AoI, and then this identifier is used to perform the corresponding matching or deployment.

Topological network information and full end-to-end application visibility should be integral to scheduling decisions, which is not the case for a fully decentralized architecture (chapter 6) or for current state-of-the-art centralized architectures (chapter 5). For example, strict latency SLOs are tightly associated with an application's deployment location relative to each client's physical location, as network traversals account for a significant fraction of each serviced request's E2E latency. Similarly, spatially proximal clients of the same coordinated application should be digitally colocated to enable essential state sharing, but the applications may span multiple µDCs, which require multi-µDC visibility to detect E2E violations. In OneEdge, we incorporate both metrics as core components of the infrastructure and, correspondingly, the scheduling process.

We use the metrics defined in section 3.4 to quantify E2E latency and spatial affinity in the proactive (*i.e.*, on-the-critical path) scheduling decisions. The *placement* module in the global domain resource scheduler selects the set of µDC(s) to launch (or reuse) the different application components of the DFG for the client request. In making this placement decision, the resource scheduler consults the aggregate state, which is the composite of the resource commitments of all the µDCs/DCs. As previously mentioned, the placement algorithm considers the network's topology to predict the latencies between application components and to determine which µDCs are equivalent (from the perspective of network location) to load-balance across them.

Due to the autonomous decision-making at the µDCs' *local manager* and the ground truth state being at the µDC, the aggregate state may be stale. Keeping the aggregate state eventually consistent [73] is a principle (section 7.1) we adopted to provide autonomy to the local domain for prompt deployment of latency-critical standalone applications. The scheduler takes decisions optimistically, assuming that the aggregate state is up to date, following the "think globally, act locally" model in distributed systems. This optimistic design also means that the aggregate state is immediately and optimistically updated in the global domain once a placement decision is made. Despite that the aggregate state is kept in an eventually consistent manner, the correctness of the placement decision for coordinated applications is ensured via the 2PC used by the transaction manager (section 7.6). The main limitation with a global domain design is that under network disconnection, the management of coordinated applications and the migration of applications become unavail-

able. However, these disconnections are expected to be intermittent, and once connectivity is reestablished, the µDC can continue normal operation. Additionally, when a µDC is unable to connect to the global domain, the client can connect to another local µDC, creating redundant paths to reach the global domain.

Finally, the resource scheduler transforms a client request into a set of concrete resource management actions. The actions, including both deployment and resource reconfigurations, specify the operations that need to be executed on the selected µDCs/DCs to achieve the target state computed by the placement algorithm. Now, we discuss how the number of resources to be used is calculated from the profiles (section 7.4.2) and a set of heuristics (section 7.4.2) used by the placement logic to improve the search of solutions when considering a geo-distributed setting.

Profiling

As part of the programming model presented in section 3.3, developers do not need to define the exact number of resources the application requires. The developers instead use an objective-based definition for deploying application instances. This design pushes the work of defining the number of resources to use to the control plane.

One additional complexity is that the programming model allows sharing application components across multiple clients to improve resource efficiency. However, due to the target applications' real-time properties, the naive sharing of resources across multiple clients using the same application instance could result in SLA violations if resources are not allocated properly. Therefore, to reduce the likelihood of an application component's overload, the scheduler should increase the component's resources each time a client shares an application instance (either by the local or global domain).

To facilitate resource sharing while respecting application SLAs, OneEdge uses offline profiling to generate the resource requirement profile (RRP), a table indexed by the app key (section 3.7) for each application component that may be shared across client requests. The RRP specifies the resource commitment needed for each application component of a DFG as a function of the number of concurrent clients. Both the *local* and *global* managers use the RRP to make allocation decisions, and it is populated in both components' instances.

The developer hints at the number of resources required to host each application component for one client. Additionally, the profiler uses a representative trace—provided by the developer—to evaluate the app with different numbers of clients sharing the DFG instance. The application then will start clients at different intervals (such that they are not

running the same section of the trace simultaneously) and test with different configurations proportional to the hint provided by the developer for one client. OneEdge has a system-wide parameter κ representing the percentage that guides the distance between successive configurations to evaluate. For example, if the resources for one client are one core and 512 MB of RAM and κ is 100%, then the profiler will first try the initial tuple (1,512), followed by (2,1024), and so on, increasing both values by 100% on each new try. In general, the next configuration to test is given by the following equation:

$$CORES = C \cdot (1 + \kappa * M/100) \tag{7.1}$$

where C is the base cores for one client given by the developer and M is the current multiple used for testing. M starts as 0 and increases up to the size of the server used for profiling (or the expected max size of the server to host that component). In general, as a heuristic, OneEdge starts the iteration of $N + 1$ clients for a given application component with the minimum viable configuration for N children, given that increasing the number of clients would never decrease the required resources.

The profiling will generate a curve with the y-axis being latency and the x-axis being the current multiple. Then, OneEdge chooses the value of M that, with 99% probability (*i.e.*, 99 percentile), will provide the required E2E latency SLA under normal configurations. This graph is generated for each application component and for each number of clients expected to be merged for the corresponding application.

While evaluating a given application component, other components are hosted in independent servers with maximum allowed resources so as not to affect the evaluated component. One important aspect of evaluation is that OneEdge obtains profiles for more configurations than the minimum necessary to determine latency requirements. This action is necessary since the minimum profile is obtained while using an optimal configuration for other configuration components. As such, this profile is not necessarily applicable when running the application in real-world scenarios. Thus some leeway may be required to find a viable end-to-end solution to the scheduling.

More complex state-of-the-art algorithms such as VideoEdge [77] reduce the search space and better define the next configurations to evaluate. The design of OneEdge's profiler is not a contribution of this dissertation, but it is a representative definition of the generation of the profiles and the selection of the configurations to be used. The profile is not in the critical path and is performed offline. When a new application is started, the amount of resources is initialized as the number of clients multiplied by the resources of

a single client. This profiling approach allows a tighter bound on the number of resources to provide the required latency, assuming that resources can amortize across clients (section 3.8.1). When there is a continuous deviation between the actual execution and the expected from the profile, the profiling is recomputed.

Guiding the placement optimization

The placement module in OneEdge is extendable and can be modified by the infrastructure provider. The placement algorithm follows the same structure as the one presented in section 4.6, and it is defined as an optimization problem with the following inputs:

- The number of clients and their locations.
- The structure and semantic requirements of the DFG.
- Infrastructure state.
- The RRP for the placement to know how many resources to use.
- Information about already deployed instances for the given application, the corresponding resources allocated, and the number of clients using each instance.

The placement output matches each application component and either a µDC or the identifier for a specific running instance of that application component. The infrastructure provider then can define how to take those inputs and select which µDCs would be the ideal locations for hosting the requested application components. The design of OneEdge's scheduler is such that it appears to be performed sequentially, so the complexity of executing the placement decision in a geo-distributed infrastructure is hidden away from the scheduler. More information is in section 7.7.2.

The design of a placement algorithm is not a contribution of this dissertation, but we present some heuristics that are appropriate for designing such an algorithm for geo-distributed resources. The following suggestions help reduce the search space and improve the convergence of optimization algorithms:

- Reuse already deployed application components if the server allows it. This decision reduces the potential effects of cold starts, reduces fragmentation, and improves resource usage efficiency by amortizing shared costs across clients.

- For application components that do not have latency constraints (or that the constraints are loose), the optimization should try to place them in a Cloud DC if the other requirements allow it (*e.g.*, bandwidth need). The intuition is that cloud resources are expected to be cheaper than geo-distributed μDC, as well as having the illusion of "infinite" capacity. This bias also reduces the resource pressure on the limited resources in the edge.

- Use geographical information to prune the search space. For example, for applications with tight latency requirements, a good starting point is the set of the closest cells (section 7.4.1). The placement algorithm can increase the search space later if this set of cells cannot host the application. Another use of geographical information is to ignore geo-distributed resources by physical limitations. For example, data cannot travel faster than 2/3 of the speed of light in the best case. Finally, if the infrastructure state contains information about the available communication mediums (*e.g.*, LoRA, 5G, WiFi), that can help define the maximum geographical distance to look for computational resources. These geographical suggestions can be encoded in the optimizer as additional constraints.

When selecting resources, the optimizer should use the equivalence of μDCs in its favor, as all the resources in a cell are equivalent and should also be encoded as so in the optimizer. Similarly, the optimizer should incorporate load balancing across this set of equivalent resources to reduce the likelihood of unnecessary violations.

Finally, all the other considerations presented in section 4.6 still apply (*e.g.*, fairness, cost, number of applications, efficiency) in the context of OneEdge, and should be deeply scrutinized when managing geo-distributed infrastructure.

7.5 Reactive policies

This section introduces the components and mechanisms required to support an agile and responsive control plane. The two mechanisms presented focus on continuously evaluating system and application metrics and reacting to them to provide the expected performance and functional requirements. The first one, monitoring, efficiently gathers those metrics and applies the corresponding policies to keep the correct operation of the applications. The second one, dynamic resource allocation, responds to local changes and allows the *local manager* to be more responsive to application variations.

7.5.1 Hierarchical Monitoring

As introduced in section 7.3.1, each local domain (*i.e.*, µDC) has its own monitoring subsystem, where the container agent obtains and periodically aggregates each locally running application components' metrics of interest (*e.g.*, per-component execution time). While the global domain's scheduler place applications so that SLOs are met, and resource utilization across the resource spectrum is balanced, continuous adaptations may be needed for various reasons. First, autonomous deployments of standalone applications at µDCs can cause utilization imbalance and increased resource pressure at individual locations. Second, client mobility can cause frequent load shifts between µDCs serving the same application and potentially change the client's AoI for coordinated applications and E2E latency for all applications. Such events can lead to application SLO violations, and correspondingly the global and *local managers* need to reevaluate their resource allocation and application placement decisions continuously from the metrics being gathered.

For local domain metrics (section 4.7), like CPU and memory usage, the aggregated statistics are conveyed to the *local manager* and handled locally. If the *local manager* detects application SLO violations, it attempts to alleviate the issue locally by allocating additional resources to the suffering application component's containers. One example of such a policy is used to modify the application components' resource allocation to improve the service provided (due to divergences from the application profile provided by the developer), and it is explained in more detail in section 7.5.2. If it is not possible to make the decision locally (*e.g.*, no local resource availability or client mobility), the *local manager* notifies the *global manager* to allow coordination across µDCs to alleviate other sources of excess latency (or other types of pressures).

For global-domain metrics (*e.g.*, E2E latency) and coordinated applications spanning multiple µDCs, per-µDC statistics must be combined to determine potential SLO violations. The *local manager* forwards the locally aggregated statistics (*e.g.*, execution times, inter-stage communication latency, queuing between stages) to a preselected *leader*, a µDC's *local manager*, which hosts some of the application DFG's components. The leader summarizes the collected statistics and sends a digest to the *global manager*'s *monitoring manager*. The global *monitoring manager* uses this information to determine if an application reconfiguration (*e.g.*, increase resource allocation at each involved µDC or migrate application stages) is necessary. If so, it generates a new request in the *request queue*. For example, in Figure 3.3, fading communication between the client and the first stage of the application would violate S_1 and would trigger a reconfiguration. There is an essential interplay

between hierarchical monitoring and scheduling (section 7.4.2). It allows decoupling the complexities of managing distributed infrastructures from applying policies for detecting SLA violations. Additionally, separating the components allows different types of distribution across the geo-distributed infrastructure. Most of the monitoring is performed locally at the µDCs, while global-domain scheduling is performed in a centralized fashion.

As a final comment about the monitoring layer, we note that the mechanisms for aggregating the data at the µDC, temporally aligning, and filtering and forwarding to the global domain manager are outside of this dissertation's scope and can be considerably complex [78]. A more in-depth analysis of the related work can be found in section 8.4. We now briefly mention the approach we used for our evaluations as a reference:

Local domain metrics: we used sliding temporal windows and percentiles for aggregating. More specifically, OneEdge keeps a sliding window of the last N measurements and calculates the metrics' required percentile.

Global domain metrics: OneEdge additionally performs a pre-aggregation step: per request grouping. Given that global-domain metrics require measuring multiple system components to obtain a final global-domain metric, we need a mechanism to combine the different measurements. When grouping, we use the leader approach previously mentioned and calculate the global-domain metric for each request processed by the framework. For example, when calculating a request's E2E latency, the leader will receive all the required latencies for computation, network transmission, and queueing after each new event is processed through the DFG. The leader then groups these values into one unique value by adding them. Then this unique metric is aggregated using the same sliding-window approach for local domain metrics.

7.5.2 Dynamic resource allocation

Situation awareness applications often display variations in workload. For example, if multiple objects are in a camera's field of view (FOV), the detector application component in Figure 2.1 may have to do more work. Similarly, if an application component is handling multiple clients, its resource requirement may increase. An example of this situation occurs if the object recognition component (Figure 2.1) reactively employs a more sophisticated algorithm based on the output of the detection component, causing the recognition component to do more work. Thus an already deployed application component may have to be dynamically provisioned with additional resources to meet the SLA requirements.

The first mechanism that the *local manager* tries when it sees unexpected latencies from the application components running locally (compared to the profile) is a gradual resource allocation increase for the container(s) hosting the target application component. The controller uses the RRP (section 7.4.2) as an application-specific guide to defining the extra resources that need to be allocated to avoid the SLO violation. First, the allocation for the affected application is increased by λ *dummy* clients, where λ is a configuration parameter (a small positive integer). For example, suppose C is the number of clients handled by the application DFG. In that case, the allocation is increased to that needed for $C + \lambda$, using RRP to identify the required resources corresponding to the new number of clients. The incremental allocation provides the agility necessary to react quickly to surges in resource needs, and it is also a tool to handle imperfect information obtained from the profile. Additional resources help reduce latency as an application component usually processes multiple clients concurrently; the extra cores would offload the computation for a subset of the clients and alleviate the component's queues.

The second mechanism is the migration of the application component instance from the µDC experiencing the load spike that caused the SLO violation to other µDCs. The *global manager* uses three pieces of information to guide this action:

- Knowledge of the cell that contains the spatially proximal µDCs to the affected µDC.
- The resource commitments at the µDCs in the cell (available from the aggregate state).
- The application requirements of the application components running.

The *global manager* using these inputs then selects application components to migrate across the cell from the most heavily loaded one to the least loaded one if it improves the overall load balance of the cell. This algorithm is quite similar to the load balancing presented in section 6.5.

7.6 Deployment and multi-domain coordination

A coordinated application may comprise several application components, which can straddle multiple µDCs due to deployment or resource availability reasons (section 3.4). Furthermore, proper application execution requires all involved components to be deployed atomically, thus necessitating cross-µDC coordination when the application spans multiple locations — all application components are required to be running simultaneously to

have guarantees for both latency and spatial-affinity SLOs. For example, even individual client mobility in the connected vehicles app would involve multi-μDC coordination since the DFG that serves that client may span multiple μDCs. As individual μDCs make autonomous deployment decisions for standalone applications, atomic modification of the distributed state is required. For this reason, we use a 2PC protocol as the starting point to enable coordinated updates of the authoritative state distributed across μDC locations.

The *transaction manager*'s role (from the global domain) is to launch the placement decisions atomically using 2PC. The term *transaction*, as it applies to this dissertation, signifies the *atomic* execution of all the associated resource management actions for a client request. This definition of the term transaction is similar to that used in systems such as LRVM [79] and Quicksilver [80], and is different from the more traditional use of the term in databases providing ACID properties [81]. The messages are exchanged between the global *transaction manager* and the corresponding *local manager*. At the end of the first phase, the transaction manager will know if all the involved μDCs have accepted the placement decision. In this case, the second phase of the transaction is to confirm the placement decision to the involved μDCs. If any of these μDCs rejects the decision in the first phase, the transaction manager sends an abort message in the second phase to all the involved locations and updates its aggregate state using the authoritative state information received from the *local managers*. The μDCs update their internal authoritative state upon receiving the abort message. After an abort, the request is re-enqueued in the *global manager*'s request queue with a higher priority. Additionally, periodic state updates from *local managers* ensure that the aggregate state does not significantly diverge from the ground truth.

As the resource scheduler updates the aggregate state after completing every request and uses the new aggregate state to process the next request, an invariant that the transaction manager should maintain is that the transactions should *appear* to be applied on the μDCs *serially*. We discuss the implementation and optimizations to preserve this invariant without compromising performance in section 7.7.

7.6.1 Re-execution of requests after aborts

There are two design options for handling a transaction abort. The first one is to let the client send a new request. The second one is for the transaction manager to re-enqueue the request for a new placement decision. The criterion for either option is that the new placement decision should use the most accurate information regarding the client (e.g.,

the up-to-date location of a mobile client) to ensure minimal impact on the SLOs for the client, and reduce potential SLOs violations. Incidentally, this criterion is relevant for even a brand new client request since there is always a latency (depending on the queue length at the central scheduler) between the submission of the request at an edge site and the execution of the placement algorithm at the central controller.

With respect to transaction aborts, the main benefit of the first option is that the new client request will have up-to-date location information. However, this incurs additional latency due to WAN traversals in the critical path (communicating the abort to the client, subsequent new request submission to an edge site, and the communication of the new request back to the central controller), unnecessarily extending the duration of potential SLO violations. Re-enqueuing the aborted request locally in the central controller avoids such WAN traversals, so long as the client information is up-to-date at the time placement decision is taken by the scheduler. Updating the client information is exactly the role of the monitoring component in the OneEdge architecture.

The latency of handling a request that is eventually aborted comprises the queueing in front of the scheduler, the execution of the scheduler, and the WAN latency to reach the μDCs. For example, in a low-load scenario, the queue is empty, the execution of the schedulers takes 2 ms, and the WAN RTT is 60 ms, which would amount to a total of 62 ms. A car moving at 100 km/h would move less than 2 meters in that period, which is less than the accuracy of regular GPS used in cars, meaning that a re-submission can use the same location for the rescheduling. On the other hand, in a scenario where the global domain is operating under high load, the scheduling latency of handling the request (due to queuing in front of the scheduler) could affect the spatial affinity SLOs. However, the monitoring subsystem of OneEdge continuously monitors the client's location after it submits a request to the control plane (at the proximal μDC); the frequency of location updates is configurable, and it is usually between 500 ms to 10 seconds (depending on the speed of client mobility). Therefore, the continuous location update of the monitoring subsystem bounds the global manager's error in estimating the client location such that it is smaller than the AoI size.

7.7 Performance optimizations

This section introduces two mechanisms to improve the performance of the baseline 2PC, focusing on reducing the possibility of transaction failures and rollbacks, and the effect of the WAN on the interactions between the *global manager* and the *local managers*.

7.7.1 Enhanced two-phase commit

To enhance the performance of coordinated application deployment, which could involve multiple µDCs, we introduce two optimizations to the traditional 2PC and dub our approach *enhanced two-phase commit*, which is a context-aware 2PC. The *global manager* uses a 2PC protocol to deploy coordinated applications spanning multiple µDCs. The traditional semantics of the 2PC protocol would abort a transaction T_i if there is a mismatch between the *global manager*'s aggregate—but potentially stale—state when T_i was generated by the resource scheduler and the µDC's authoritative state when the µDC's *local manager* process T_i.

Using 2PC semantics as the starting point simplifies the *global manager*'s state management since the authoritative state is held in the respective geo-distributed µDC. Furthermore, to avoid unnecessary aborts, OneEdge leverages the observation that a transaction need not abort as long as the sum of the requested resources by transaction and currently reserved resources do not exceed the µDC 's resource capacity (Omega [38] exploits a similar idea in a datacenter setting). When such conditions are met, instead of aborting the transaction, the µDC's *local manager* updates the authoritative state with the transaction's allocation request during phase one of the protocol and informs the *global manager* of the actual µDC resource commitments to update the aggregate state (which includes the deployment of the new application component). The key difference between baseline and enhanced 2PC is the semantic redefinition of what constitutes a state conflict (i.e., aggregate state and µDC's states do not need to match, but sufficient resources must be available). The goal of the coordination between the local and global domain in enhanced 2PC is twofold: to maintain the capacity invariant and to deploy all the components in the application atomically. Enhanced 2PC subsumes the validation of the capacity invariant inside the first phase of the 2PC instead of building it on top of the 2PC, as is done in traditional databases, thus accommodating requests that would otherwise have caused violations due to state mismatch between the local and global domains.

One additional complexity that arises with this optimization is that clients' original requests are objective-based and that the programming model allows the merging of application components. These two characteristics cause that going from *n* clients to *n+1* clients may not perform the same resource changes as going from *n+1* to *n+2*, given that the delta of additional resources to be assigned in one condition may be different from the other scenarios. This scenario can only happen for deflections of standalone application requests. For this, a second change is required in which the resource management requests

that compose a transaction are not for raw resources but for launching or modifying an application component with respect to the number of clients being served. The µDC's *local manager* uses the semantic knowledge encoded in RRP (section 7.4.2) to allocate resources commensurate to the actual change in the number of clients currently being served by that location.

The second optimization to conventional 2PC reduces the WAN round-trip per transaction. Nominally, the application execution after a coordinated application deployment can proceed if the *global manager* affirms in the second phase that the deployment request was successful in all the participating µDCs at the end of the first phase. However, this 2PC protocol would entail two full WAN round trips. Instead, we propose an optimization that reduces the latency on the critical path from two WAN traversals to one. In the first phase, a µDC replying affirmatively to a deployment request also reserves the requested resources. If the *global manager* receives affirmative responses from all affected µDCs, it notifies the client in parallel with the execution of the second phase. Thus, the WAN latency for the second phase can be overlapped, as the µDC can start receiving actual data plane actions from the client ahead of the second phase's completion. If the transaction is aborted, the second phase frees each µDC's reserved resources. For example, a deployment request to allocate a new container would result in the completion of the actual allocation, the initialization, and the control plane setup for data communication with other stages of the application pipeline in anticipation of the successful second-phase message from the *global manager*.

Similarly, an extension to the second optimization is for a request that increases the resource allocation for an existing application container to be carried out proactively in the first phase. Thus, the control plane actions needed for the application component's start are completed ahead of the second-phase commit message. The only exception is decreased resource allocations, performed after the second phase to avoid negatively impacting an application component.

The main drawback of the second optimization is that OneEdge may reserve resources ahead of time and reduce the likelihood of a local request (*i.e.*, from the µDC) succeeding when trying to obtain resources from the potentially scarce capacity of the µDC. Depending on the priority of the different types of requests (*i.e.*, standalone vs. coordinated) and across different applications, it may be useful to consider deactivating this third optimization. However, it provides an important reduction in the effect of WAN on global decisions and should be evaluated by the infrastructure provider when configuring the control plane.

7.7.2 Transaction pipelining

It is reasonable to expect geo-spatial diversity of µDCs across successive client requests arriving at the *global manager*. Successive transactions affecting *disjoint* sets of µDC are independent. The transaction manager should attempt to execute independent transactions in parallel to exploit this opportunity. However, for the correct operation of the resource scheduler, transactions should appear to be executed serially by the transaction manager. This invariant has to be guaranteed by the transaction manager while exploiting the opportunity for executing *independent* transactions concurrently.

One way of exploiting parallelism and preserving the ordering invariant is to enforce ordering at the destination µDCs. The following conditions should be met at the target µDC to ensure that transaction order is preserved while executing transactions (which may or may not be independent of each other) in parallel:

1. The µDC should process successive transactions that affect the same µDC *in the scheduler's order of generation*.

2. A transaction abort should correctly restore the authoritative state at the µDC before that µDC processes subsequent transactions.

Further, on the *global manager* side, an aborted transaction should roll back the aggregate state to free up the resources committed for that transaction.

Design implementation. The transaction manager in the global controller is separated into two entities: the *pending commands* data structure and the *transaction executor*, as shown in Figure 7.3. This split allows the scheduler to update the *aggregate state* optimistically. The resource scheduler places new transactions in the *pending commands* data structure. The *transaction executor* takes these transactions and launches them on the target µDCs. This decoupling means that the resource scheduler can continue processing subsequent requests without waiting for the previous request to complete successfully. In addition, this design allows the resource scheduler to be completely impervious to aborted transactions, separating the concerns of the placement and SLO support from those of managing executions across a geo-distributed infrastructure.

The transaction manager maintains a *dependency graph* for each transaction. We define a *dependency* as an overlap in the set of µDCs modified by two transactions. A transaction T_i depends on T_j (denoted by $T_j \rightarrow T_i$) if the following conditions hold:

- T_j was added to *pending commands* before T_i.

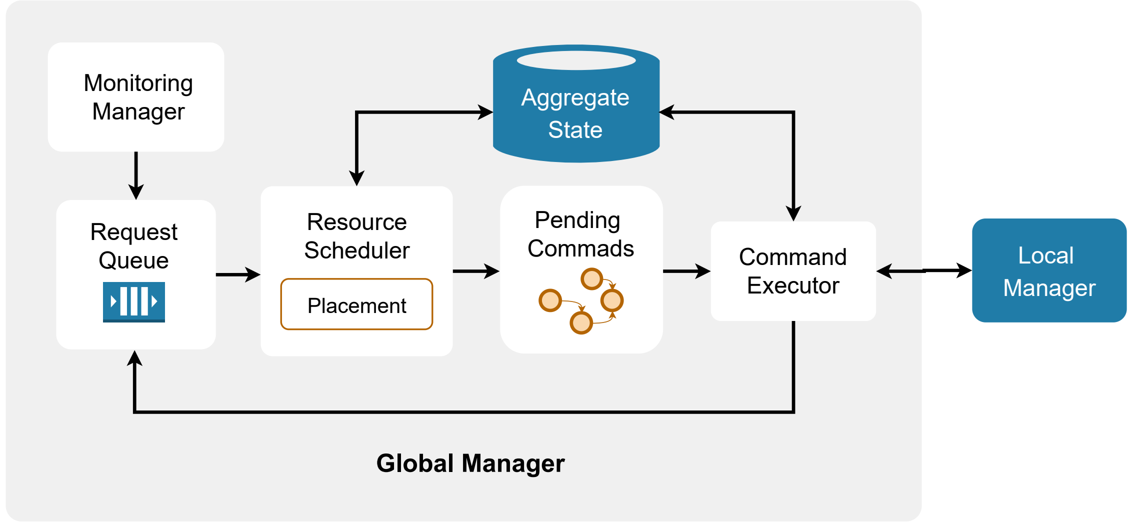

Figure 7.3: The *transaction manager* is split into two subcomponents: *pending commands* and *command executor*. The *pending commands* maintains a directed-acyclic graph with the resource management actions (i.e., transactions) defined by the scheduler until the *command executor* completes them. The *command executor* is the entity in charge of coordinating with each of the associated local managers for each transaction.

- T_j has not been completed, and T_j creation time precedes T_i's.
- $\mu DCs(T_i) \cap \mu DCs(T_j) \neq \phi$.
- T_i used the *aggregate state* assuming the successful completion of T_j for its placement decision.

We denote $D(T_i)$ to represent the dependency set of transactions T_j for which $T_j \rightarrow T_i$. Similarly, we denote $AD(T_j)$ to denote anti-dependency, *i.e.*, the set of transactions T_i such that $T_j \rightarrow T_i$. Every transaction T_i sent to a µDC contains $D(T_i)$ and $AD(T_i)$. Thus the *pending commands* data structure is a directed acyclic graph, where each vertex represents a transaction, and a directed edge exists from T_j to T_i iff $T_j \rightarrow T_i$. A transaction T_i is eligible to be processed at each affected µDC so long as all the transactions in its dependency set $D(T_i)$ have been completed.

The eligibility condition could be enforced conservatively by the *transaction executor* at the transaction's launch time or optimistically at the destination µDC by the µDC's *local manager*. OneEdge takes the optimistic approach, enforcing the condition at each µDC's *local manager*, which processes the request queue of incoming transactions to that µDC.

µDCs are made aware of inter-transaction dependencies with special metadata contained with each received transaction. For every new transaction T_i the transaction manager sends to a given µDC, T_i's metadata indicates the $T_j \rightarrow T_i$ dependence.

The µDC will not process T_i unless it has already received and processed T_j. The completion of T_j will trigger the deletion of all the incoming edges from the transactions in $AD(T_j)$, possibly making some of them eligible for processing. If T_j is aborted, the *aggregate state* is rolled back accordingly. Further, this abort will trigger a cascading rollback of the pending transactions that transitively depend on T_j (*i.e.*, starting from the members of $AD(T_j)$). These aborted transactions will result in a re-submission of the associated control plane requests to the global scheduler's request queue. However, aborts are uncommon because, per our *enhanced 2PC* protocol, they only occur if the µDC's resources have been depleted (and not due to a mere mismatch between the *global manager*'s state and the µDC's state of resource availability), as evidenced in our evaluation (section 7.10.2).

To prevent queue buildup at the destination µDCs, we use a *windowing* technique (similar to the TCP protocol), limiting the maximum number of outstanding transactions sent to a given µDC. The limit ensures that a µDC is not overloaded. Upon completing a transaction, the next pending transaction of a µDC (if any) can be launched to keep the window full. In addition, the pipelining of operations helps to hide the effects of WAN due to the dependence between successive decisions and its associated state modifications.

7.8 Fault Tolerance

Fault tolerance for the global domain manager is provided using standard mechanisms. We assume that the global manager runs in a robust environment (*e.g.*, Cloud datacenter). While the server that hosts the global manager may fail, it is improbable for the entire datacenter to go down. Therefore, the fault tolerance approach is to have a secondary instance of the global manager running in parallel on another server. All the pertinent state involved in the primary workflow (section 7.3) is replicated, including in-flight transactions. On a primary failure, the secondary takes over and rolls back to an aggregate state comprising only complete transactions, issuing aborts for all the in-flight transactions.

7.9 Implementation

OneEdge is implemented in C++11 on Ubuntu 18.04. Each application component is dynamically linked into a base OneEdge container image built using the Docker framework,

similar to how it was implemented in the previous chapter with Foglets (chapter 6). In addition, we use MongoDB to store the *aggregate state*, and ZMQ for communication among the system's distributed components.

7.10 Evaluations

The following are the hypothesis that guided the evaluation in this chapter:

1. The optimizations, transaction pipelining and enhanced 2PC (section 7.7), are successful in improving OneEdge's performance (section 7.10.2).

2. OneEdge's hybrid architecture allows it to achieve lower-latency placement decisions compared to a centralized architecture for standalone requests (section 7.10.2).

3. OneEdge achieves a good compromise between deployment latency for standalone requests and load balance across equivalent µDCs in a cell (section 7.10.2).

4. OneEdge is comparable to a centralized control plane at meeting both application types' SLOs, while considerably reducing deployment latency for standalone requests (section 7.10.3).

Section 7.10.2 presents microbenchmarks to verify the first three hypotheses. Additionally, section 7.10.3 presents an E2E evaluation to verify the fourth hypothesis. We use as a baseline a variation of OneEdge where the *local managers* deflect *all* incoming requests at a µDCs to the *global manager*. The resulting baseline is functionally equivalent to KubeEdge [46], with two improvements from OneEdge architecture: the support for pipelined control plane actions described in section 7.7 and for satisfying the applications' E2E latency constraints, which vanilla KubeEdge cannot currently provide.

7.10.1 Experimental platform

For both evaluations (*i.e.*, microbenchmarks and E2E), we emulate a geo-distributed computational infrastructure with µDCs in multiple metropolitan areas. We use resources in five different Azure regions: WestUS, WestUS 2, CentralUS, South Central, and East US.

We host the *global manager* in the East US region. Each of the remaining four regions is used to emulate a different metropolitan area. In each Azure region, we create multiple VMs, and each VM is designated as a µDC located within that metropolitan area. Each of the VMs is of type "Standard D16s_v4" (with 16 vcpus and 64 GiB memory). We emulate

Table 7.1: Summary of parameters for microbenchmarks of OneEdge.

Parameter	Value
Container Startup/Update	583 ms/25 ms
One-way WAN latency	20 ms
Window Size	100
Resource Scheduling Latency	2 ms
Modeled per-µDC resource capacity	4096 cores, 8 TB memory
Per-request resource allocation	1 core, 512 MB memory
Coordinated application exponential β range	10–100 s
Coordinated application Poisson λ range (per µDC)	1–8 s^{-1}
Standalone application exponential β range	100–300 s
Standalone application Poisson λ range (per µDC)	2–25 s^{-1}

clients in each metropolitan area and are hosted within the corresponding region. Clients move only within their associated metropolitan area. In other words, clients move across their area's µDCs but do not cross to other metropolitan areas.

7.10.2 Microbenchmarks

In this section, we stress-test OneEdge by executing all the control plane actions without any application running or any actual resource allocations. The control plane does keep the accounting of the resources used and all the coordination mechanisms; the only step not performed is starting and executing the containers. To stress-test at scale, we emulate µDCs and parameterize the container runtime within each µDC. Further, we generate the client workload presented to OneEdge to drive the controlled experiments.

Experimental Setup

Control-plane Parameters. OneEdge consists of one *global manager*, multiple *µDCs* with their associated *local manager*, and multiple *clients* (Figure 7.1). We use a simplified placement algorithm to evaluate our performance optimizations, as a faster placement logic would stress more the transaction manager. The manager's resource scheduler uses two common heuristics: round-robin placement across µDCs (*i.e.*, the same metropolis) to improve allocation balance, and collocation of an application's DFG components on the same µDC, if capacity allows, to improve application E2E latency. Additionally, we set resource scheduling latency to 2 ms. This setting is similar to the lower range for resource scheduling latency in Kubernetes [82] and matches one of the best-case performances of

our resource scheduler implementation used in the end-to-end study (section 7.10.3).

Emulated µDCs. We use a capacity of 32 servers with 128 cores and 256GB of DRAM per server to model the size of a µDC. The modeled per-µDC capacity is only for book-keeping during the microbenchmark experiments, and resources are not allocated in the microbenchmarks. In addition, we use one Azure VM to host each µDC's *local manager*, with no actual deployments of components.

Container Runtime. We measure the Docker container runtime used by the *local manager* implementation (Figure 7.1) to parameterize the response times associated with the execution of application components on a µDC. The mean measured container deployment time (consisting of a simple application and its container agent library) is 583 ms and a standard deviation of 143ms. The mean measured time for updating an already deployed container's resource allocation (CPU-set and memory limit [83]) is 25 ms and a standard deviation of 4ms. We use these results to emulate the µDC's reaction time upon every microbenchmark deployment request.

Workload Characterization. We derive a synthetic workload with a combination of deployment requests for both coordinated and standalone applications. We create the synthetic workload from the cellular mobility of cars in San Francisco (SF), using the SF cabs dataset [84] and locations of cellular towers in SF [85]. We group the cellular towers with k-means into 32 clusters and select each generated cluster's centroid as a µDC's location. The clients send allocation requests to the geographically closest µDC. Hence client-µDC communication does not incur WAN latency in our microbenchmarks. Due to the mobility of the dataset, each client's closest µDC will change during the evaluation. For coordinated application clients, this will trigger migration requests. For standalone application clients, it will send an allocation request to the new closest µDC.

The SF cab dataset daily taxi count represents only a small subset of the city's expected fleet. To more closely match the mobility of cars in San Francisco, we overlay 23 days' worth of data in the SF cabs dataset to increase the number of simultaneously active cars. Each µDC 's request arrival is modeled using a Poisson distribution. Similarly, each client's connection duration to a given µDC is modeled using an exponential distribution. Finally, we parametrize both of these distributions with the ranges of client inter-arrival and connection duration extracted from our enlarged taxi cab dataset for both coordinated and standalone applications. Table 7.1 shows the parameters used in the microbenchmarks. To conduct larger-scale experiments with multiple metropolitan areas, we replicate the geo-distributed infrastructure (using additional Azure Regions) and use the above SF workload

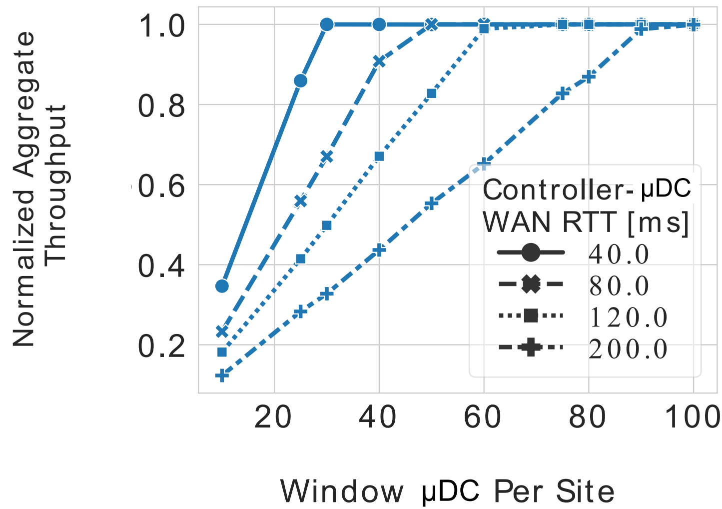

Figure 7.4: Impact of Pipelining Optimization on Aggregate Throughput.

for every additional metropolitan area we model.

Control Plane Configurations:

We use four configurations for the evaluations: the Centralized equivalent and three configurations of OneEdge with different deflection thresholds for standalone requests: 0.5, 0.75, and 1.0, while the deflection percentage is set to 100%.

Evaluation of OneEdge's Optimizations

Next, we evaluate the two optimizations discussed in section 7.7: transaction pipelining and enhanced 2PC, using a single µDC.

Transaction Pipelining. The windowing mechanism in the *transaction executor* (section 7.7) aims at reducing the effect of the WAN RTT between the *global manager* and the µDC on the throughput of scheduling requests. Therefore, the optimal window size choice varies with respect to the WAN RTT.

To understand the effect of the window size on OneEdge's throughput, we perform an experiment with only coordinated application requests, where the request generation rate is chosen to correspond with the maximum throughput achievable by the *global manager* without queuing delays (*i.e.*, comparable to not having a WAN RTT of zero nor transaction aborts). Figure 7.4 plots the effect of window size on the aggregate throughput (normalized to the maximum throughput achievable) for different WAN RTT configurations. As expected, the optimal window size required to maximize throughput grows as a function

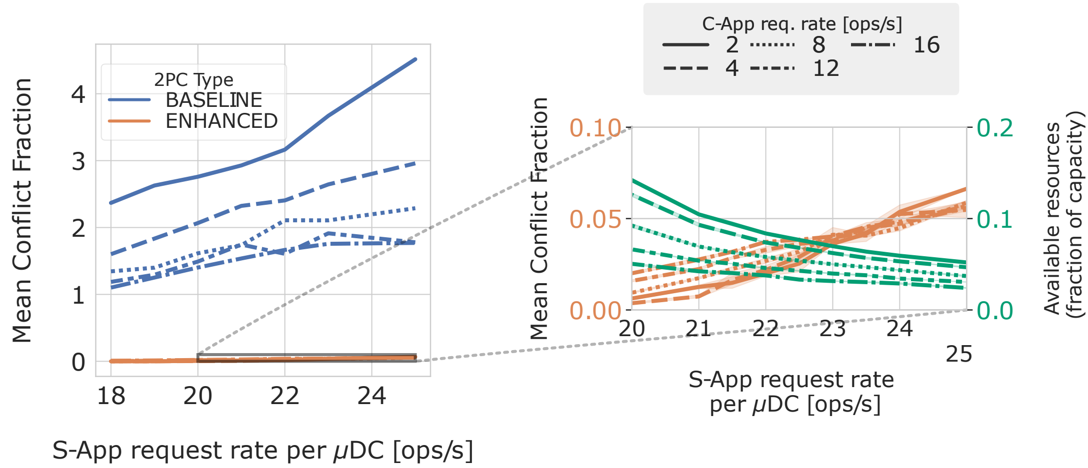

Figure 7.5: MCF of baseline and enhanced 2PC for constrained resources at a µDC and typical coordinated and standalone application request rates from the SF cabs dataset (Table 7.1). The blow-up shows the increase in MCF and the µDC's remaining available resources for enhanced 2PC at higher request rates.

of WAN RTT. For example, a WAN RTT of 40 ms requires a minimum window size of 50 transactions to reach its maximum throughput. This result emphasizes the need to perform multiple placement requests concurrently to mitigate the negative impact of WAN RTT from the *global manager* to the µDC. Therefore, we chose a conservative window size of 100 for all the remaining microbenchmark experiments based on these results.

Enhanced 2PC Protocol. This optimization goal is to reduce avoidable rollbacks of scheduling requests due to state mismatch between the *global manager* and the µDCs' *local managers* via state reconciliation. We use the metric mean conflict fraction (MCF) to evaluate its effectiveness, defined as *the average number of conflicts per successful transaction* [22]. A zero value represents no conflict, while a non-zero value indicates the number of aborts for each successful transaction.

For this microbenchmark, we use a combination of coordinated and standalone application requests and disallow *deflection* to fully control which requests are handled at the *global manager* as opposed to a given *local manager*. Given that state reconciliation is harder when the resource commitment at µDCs is high, we focus the evaluations on such a scenario. Therefore, the evaluation uses the higher arrival rate ranges from Table 7.1: it varies the coordinated application and standalone application request arrival rates between 2–16 and 15–25 requests per second, respectively, while keeping the coordinated application and standalone application client durations fixed at 50 and 200 seconds, respectively.

Figure 7.5 shows the MCF for the enhanced 2PC compared against baseline 2PC over the range of request arrival rates presented before. The MCF of baseline 2PC is consistently higher than the enhanced 2PC, which only becomes non-zero at high arrival rates when the resource commitment at the µDC is sufficiently high to cause transaction failures due to capacity overcommitment. For example, at a standalone application rate of 20 req/s, the µDC resource commitment is 88–90% of its total capacity, and it is only from this point on the MCF increases for enhanced 2PC. Figure 7.5's inset plot is a blow-up of the enhanced 2PC results to show the increase in MCF with increasing request rates. Even at an observed capacity of 95% (corresponding to the largest standalone application arrival rate shown in the graph), the MCF for enhanced 2PC is an order of magnitude lower than baseline 2PC. On realistic deployments with multiple µDCs, the *global manager* can further reduce the probability of capacity-caused conflicts by avoiding scheduling new requests on µDCs with resource commitments over a threshold (*e.g.*, 80%).

The higher the MCF, the higher the probability of failure, hurting the latency of coordinated application requests because their successful execution requires repeated scheduling attempts across the WAN. The MCF trends can be used to extrapolate the probability of a failure for applications deployed across n µDCs: the probability of failure is $1-(1-f)^n$, where f is the probability of transaction failure on a single µDC, which equals $MCF/(1 + MCF)$. A higher MCF increases the likelihood of failure, which means that the enhanced 2PC's positive effect is multiplicative in the multi-µDC scenario.

In the inset of Figure 7.5, there is an additional trend that deserves a detailed explanation. For enhanced 2PC, higher rates of C-app requests (*i.e.*, 8 and 16 op/s) reduce the MCF slope when compared to lower C-app rates (*i.e.*, 2 and 4 op/s). However, the starting non-zero MCF value for lower C-app rates is smaller than for higher C-app request rates. The diverging initial MCF values are due to the difference in the available capacity and the frequency of aggregate state updates at various C-app and S-app rate configurations. For example, for S-app request rates between 20 and 22 op/s, lower C-app rates have more available resources than higher C-app rates, reducing the probability of capacity violation, and giving a lower MCF value. On the other hand, increasing S-App request rates makes the difference in available capacity across different C-App request rates less pronounced. Hence, the role of more frequent updates to the aggregate state with higher C-App request rates becomes more prominent - thereby explaining the lower slope of the MCF trend with higher C-App request rates.

The windowing and enhanced 2PC experiments validate our first hypothesis regarding

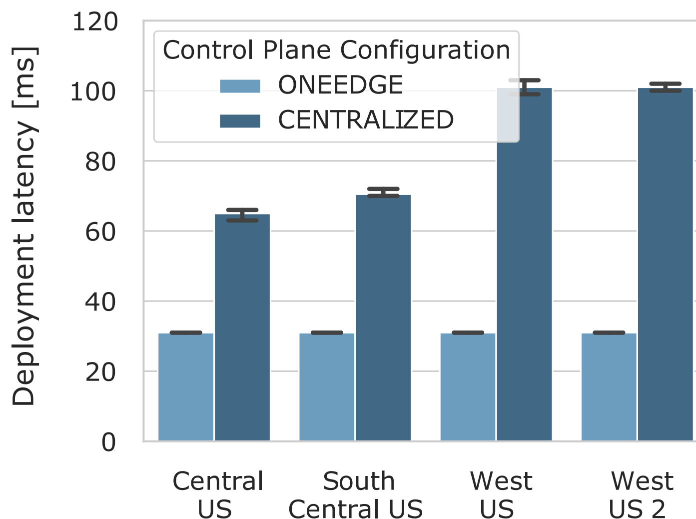

Figure 7.6: Standalone deployment latency: comparison between centralized and OneEdge's control planes.

the effectiveness of OneEdge's optimizations in improving performance.

Control Plane Effect on Standalone Applications

Next, we evaluate OneEdge's performance improvement over a centralized control plane. As previously mentioned, the centralized baseline is comparable to KubeEdge [46] concerning the overhead of control plane actions. Therefore, we evaluate the latency per standalone application deployment request for this microbenchmark. The experiment uses a µDC in each of the four metropolitan regions (section 7.10.1). Naturally, µDCs hosted in different Azure regions perceive different WAN latencies to reach the *global manager*. In this microbenchmark, deployment requests are created from each of the µDCs with the parameters in Table 7.1. Deflection is turned off to control where each request is processed.

Figure 7.6 displays the deployment latency of standalone application requests from each of the four metropolitan areas. The scenarios chosen had a low load to avoid queue buildup in the scheduling entity. Additionally, we divide the requests based on their originating metropolitan area. Centralized incurs higher deployment latency than OneEdge and is higher for µDCs further away from the *global manager*. In contrast, OneEdge incurs a constant low latency irrespective of the WAN latency between the *local manager* and the *global manager*, as it depends only on the container's allocation update latency (25 ms as per Table 7.1). Centralized incurs a high latency for all the standalone application

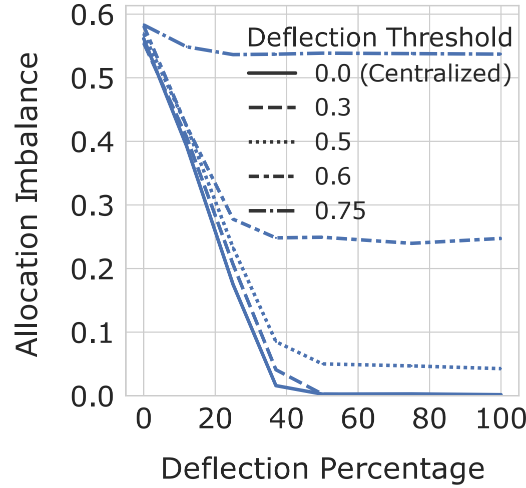

Figure 7.7: Allocation imbalance for standalone application request handling at proximal µDC vs. at the *global manager*. Shown results are for the cell emulated in the West US Azure region.

rates evaluated. The deployment latencies for coordinated applications requests are similar for centralized and OneEdge since both incur the WAN RTT; we do not plot them for brevity. However, OneEdge's latency values tend to have lower variance due to the reduced standalone application requests interference.

These latency results corroborate the second hypothesis regarding the advantage of OneEdge over a centralized control plane in terms of deployment latency for standalone application requests.

Latency Versus Load-balance Trade-off

OneEdge's deflection mechanism (section 7.4.1) allows the *global manager* to load-balance standalone applications across µDCs that are part of the same cell (*i.e.*, are equivalent in providing the latency requirements of the requesting client). We create a microbenchmark that evaluates the trade-off between achieving low latency for standalone application requests and the desired property of resource allocation balance across a cell. The metric used is *allocation imbalance*, defined as the *difference between the highest and lowest resource commitments among the µDCs in a cell at a given time* [22]. An allocation imbalance with a value of zero indicates perfect load balance.

The evaluation is performed with eight µDCs, all located in the same metropolitan area. Each µDC is capable of meeting the E2E latency requirements of all the emulated client requests within that geographical region. We calculate the allocation imbalance for all the

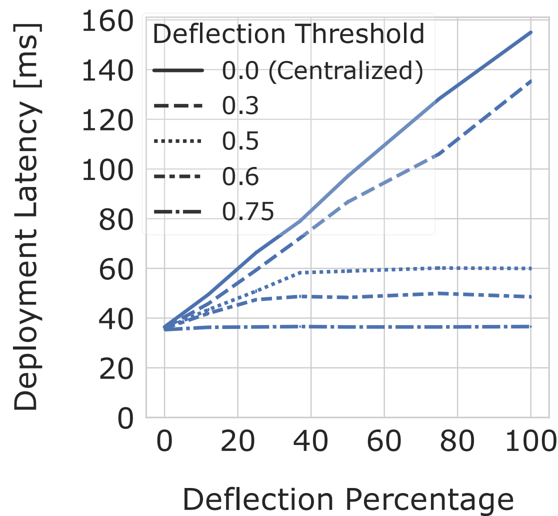

Figure 7.8: Deployment latency for standalone application request handling at proximal µDC vs. at the *global manager*. Shown results are for the cell emulated in the West US Azure region.

µDCs. When the *global manager* receives a deflected request, its placement algorithm selects the µDC with the lowest resource commitment towards a more evenly balanced load.

The workload consists of only standalone application requests, which are purposefully skewed such that if all the requests were handled locally (*i.e.*, no deflection), 50% of the µDCs in each cell would have 80% of their resources committed, while the remaining 50% of the µDCs would have only 20% of their resources committed on an average. In other words, this workload without deflection results in an allocation imbalance of ∼0.6.

We consider a cell composed of eight µDCs emulated in the West US Azure region, but similar trends hold for the other Azure regions. Figures 7.7 and 7.8 display the allocation imbalance when applying deflection. The figures highlight the trade-off between deployment latency and allocation imbalance for different configurations of the deflection *threshold* and *percentage*. For example, increasing the deflection percentage for a given threshold results in a better allocation balance (Figure 7.7) at the expense of higher request completion latency (Figure 7.8) and a higher load on the *global manager*.

An interesting result is that for a given deflection percentage, lowering the deflection threshold, which would result in increasing the number of deflections, does not result in a proportionate decrease in allocation imbalance. For example, in Figure 7.7, a significant reduction happens in allocation imbalance between thresholds of 0.5 and 0.6. In contrast,

the reduction in allocation imbalance between 0.3 and 0.0 is much smaller. This result is intuitive because the 0.5 threshold is close to the average resource commitment for the evaluated group of µDCs.

These results support our third hypothesis regarding the use of deflection for the latency / load-balance trade-off. Further, they suggest a simple policy for defining the deflection threshold, namely, the average resource commitment across equivalent µDCs. This policy would be easier to implement than striving for an optimal trade-off, which would require obtaining the lifetime of the deployed applications on the µDCs that may not be readily available.

7.10.3 End-to-end evaluations

The microbenchmarks evaluated OneEdge by stress-testing the control plane without real allocation of µDC resources or execution of application components. This section describes an E2E evaluation using the experimental platform in section 7.10.3 and OneEdge's full implementation for running exemplar situation awareness application mockups. The E2E evaluations compare OneEdge to configurations that are architecturally comparable to fully centralized and fully decentralized control planes. The objective of the E2E evaluation is to demonstrate our last hypothesis: OneEdge's ability to provide both low latency for standalone applications and meet application SLOs expressed as latency bounds and spatial affinity for coordinated applications.

Experimental Setup

Applications.

We use two applications to perform our end-to-end evaluations:

Drone: it is an instance of a standalone application. It is based on the work of Samira Haya et al. [27] and Alex Zihao Zhu et al. [28]. It uses inputs from a camera and an inertial measurement unit (IMU) to determine the pose and location of the drone, using a type of extended Kalman-filter algorithm. The drone application's pipeline comprises two stages:

1. Feature tracking and detection from the IMU and cameras inputs.
2. Pose state estimation from the features extracted (update).

For our evaluations, we use a dataset generated using the ROS [86] framework for both the inputs of the camera and IMU [87]. To model the mobility of the drones

(each drone operates independently), we use the San Francisco cab dataset [84], associating individual cab mobility with that of a drone. This drone application (S-app for short) is run with the above synthetic dataset and mobility data for a mockup of the standalone application for our E2E evaluation studies.

View-Fuse: it is an instance of a coordinated application, and it is based on Zijian Zhang et al. [88]. This application fuses the objects detected by multiple autonomous vehicles from their respective fields of view to create an expanded world sub-regional view (Figure 3.3), which is then sent back to the vehicles in the same geographical locale to improve collision avoidance decisions. To create a mockup of this application for our evaluation purposes, we first created a dataset using Carla [89]. Specifically, we used 80+ cars driving through the most complex map directly available in Carla (called Town3). A 15-minute Carla simulation produces a spatio-temporal dataset consisting of object detections by individual vehicles. This dataset is then used as the input to a multi-car fusion application (C-app for short) for a mockup of the coordinated application for our evaluation studies.

Mixed Workload Creation.

For the E2E evaluation, we created a combined workload consisting of both standalone and coordinated applications utilizing the control plane simultaneously. The maps are different for the two applications in the above data collection. However, the only purpose of the map is to assign a spatial location for a client relative to others in the same application. Therefore, to unify the data corresponding to each of the two maps, we shrunk the larger map (San Francisco city) so that its four corners are aligned with Carla's Town3 map. The implication from the application point of view is that the drones appear to move slower than in the original dataset, as usually, drones move slower than cars on average.

Control Plane Configurations.

We use four configurations for the evaluations: the Centralized equivalent and three configurations of OneEdge with different deflection thresholds for standalone requests: 0.5, 0.75, and 1.0, while the deflection percentage is set to 100%.

Emulated µDCs.

To emulate a Metropolitan area, we map each area to an independent Azure Region. Then, each µDC is represented by a VM in the corresponding Azure region. Each VM in Azure has 16 vcpus with 64 GiB of memory, as previously mentioned in section 7.10.1. In contrast to the microbenchmarks, the containers are actually hosted and executed, and the application DFGs execute on them for the E2E evaluation.

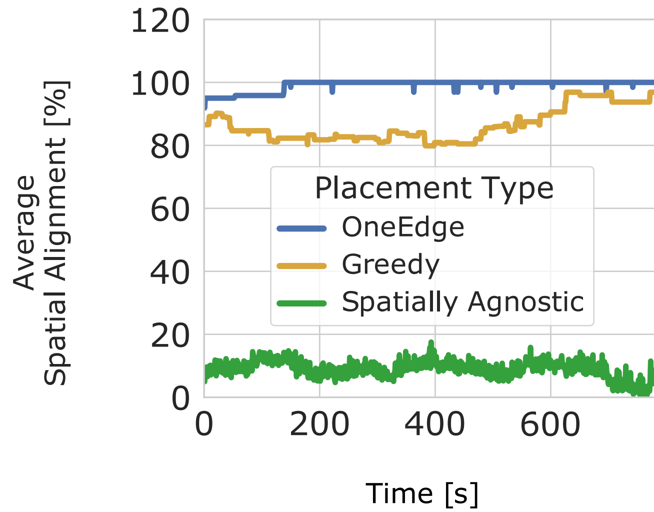

Figure 7.9: End-to-end Evaluation of Hybrid architecture: Spatial alignment for the coordinated application: OneEdge vs. greedy placement.

Evaluation SLO Metrics.

In addition to the control plane-specific metrics such as deployment latency and spatial alignment, we also measure the SLO violations for all deployed application components. We define the latency SLO as follows. First, the latency bounds for the C-app are 10 ms for the first level (*i.e.*, the sub-regional view), while the expanded sub-region view component has a latency bound of 100 ms, numbers aligned with prior work on autonomous cars [90]. Second, the latency bound for the S-app is 12 ms for the first level (*i.e.*, feature tracking) and 50 ms for the second level (*i.e.*, update). Any perceived latency exceeding these bounds is considered an SLO violation.

Modeling Network Delays.

We use two mechanisms to model network latency. First, we use a function based on the geodesic distance for µDCs, where latency is proportional to how far the µDCs are from each other. This mechanism assumes direct communication among the µDCs locations, *a la* VaporIO [17]). Second, we divide composed maps into regions where each represents a cell's coverage area for mobile clients. When a client moves and is no longer in the same region as the µDC's cell, an overhead of 15 ms is added, representing the penalty of hand-overs and the communication through the back-haul infrastructure.

Analysis of Results

Figure 7.9 shows the obtained spatial alignment for the *C-app* application. The graph displays a representative time window for the *C-app*'s execution in one of the metropolitan areas. We created 48 AoIs to divide the metropolitan area and present the mean spatial alignment (calculated for each AoI as per Equation (3.5)). The plot includes both *Greedy* and *Spatially Agnostic* placements as reference points. The *Greedy* placement represents a fully decentralized design similar to Foglets [23] in chapter 6. Optimal placement would achieve 100% spatial alignment.

Suboptimal spatial alignment indicates that the fused results returned to the cars by the sub-regional View in Figure 3.3 are incomplete, resulting in lower-fidelity decision-making by the vehicles. Greedy selects the closest µDC every time a car requests to connect to the View-Fuse *C-app*. Greedy is an idealized approximation of any real greedy implementation: it is computed offline and does not account for additional latencies incurred by migration or deployment latencies. Spatially Agnostic places each request on any µDC with available resources without considering the client's location, as would be the case if we tried to use Kubernetes scheduler out of the box.

Figure 7.9 highlights that *OneEdge* achieves near-perfect spatial alignment, outperforming Greedy with downward spikes attributed to migration latency that causes the attained spatial alignment to lag behind the ground truth of the vehicles' spatial affinity. The huge gap with Spatially Agnostic indicates the significance of spatial affinity in the control plane's placement decisions.

For both Figure 7.10 and Figure 7.11, we set the number of clients of *C-app* (vehicles) to 72 and sweep the number of clients using the standalone (drones). In both evaluations, a deflection threshold of 1.0 is architecturally comparable to a fully decentralized approach similar to Foglets [23].

Figure 7.10 displays the mean deployment latency for standalone applications while excluding outliers from the calculation (*e.g.*, initial cold starts). When there is sufficient µDC capacity and OneEdge handles all deployments locally (deflection threshold 1.0), the achieved deployment latency compared to centralized is more than $3\times$ lower. As more requests are deflected, the deployment latency gap between OneEdge and the centralized control plane is reduced. In other words, a higher deflection threshold in OneEdge yields lower standalone deployment latency. In the evaluations, an increase in the number of drones also raises the probability of reaching a µDC's deflection threshold, leading to more deflections and thus higher latencies for *S-app applications*.

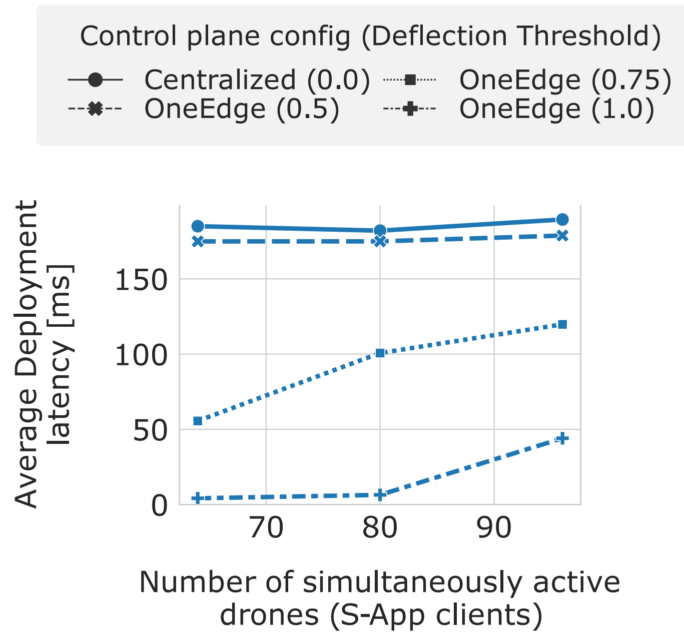

Figure 7.10: End-to-end evaluation of the hybrid architecture: deployment latency for standalone applications.

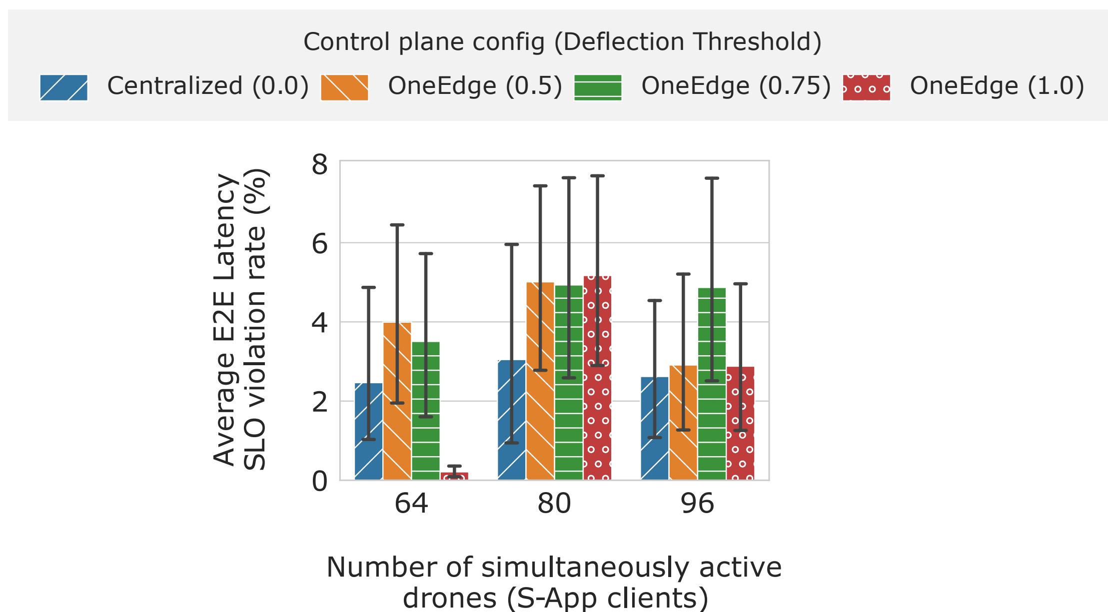

Figure 7.11: End-to-end evaluation of hybrid architecture: end-to-end latency SLO violations detected for the coordinated application.

Finally, Figure 7.11 shows the arithmetic mean percentage of SLO violations for the sub-region view application component of the *C-app* application under the different control plane configurations. Similar to Figure 7.10, the violations are presented for a changing number of active drones. All three OneEdge configurations display similar trends for SLO violation rates as centralized while delivering much better deployment latency for S-app applications.

7.10.4 Discussion: extending OneEdge evaluations to higher request rates and scalability limitations

The evaluations in the previous sections use traces extracted from real taxi cabs mobility datasets and simulated environments (*i.e.*, Carla simulator), and execute on infrastructure locations modeled after the San Francisco cell tower locations. The mobility traces and infrastructure properties then defined the request rates that would be expected at each μDC for both coordinated and standalone applications, as well as the duration of the corresponding application instance in a given locale. A change of either distribution —request rates or resource allocation duration— could affect the absolute numbers in the evaluations and shift the bottlenecks to a different component of OneEdge. Two main evaluations would be affected: conflict fraction and throughput.

The MCF evaluation in Figure 7.5 sweeps over request rates for standalone applications of up to 25 operations per second per μDC. Although this number seems relatively low, the rate is not limited by the control plane scalability but by the capacity to host applications in a given μDC. The MCF evaluation can only be measured up to the point when the available capacity (green line in the inset of Figure 7.5) drops to zero. If we increase the capacity of the servers from what was shown in Table 7.1, the evaluations can be generated for higher requests of coordinated and standalone applications. A higher standalone application rate would more heavily impact traditional 2PC as it increases the likelihood of a mismatch further; on the other hand, it does not affect enhanced 2PC until capacity is higher than 80%. A higher coordinated application rate would improve the relative MCF slope for both traditional and enhanced 2PC as the aggregate state would be updated more frequently. For traditional 2PC, the ratio between coordinated and standalone rates is what defines the MCF. On the other hand, for enhanced 2PC, the available resource capacity would still be the main driver for MCF at higher coordinated application rates. However, there will be a point where the bottleneck will be the schedulers in both global and local domains.

The latency of a scheduler is usually in the order of low milliseconds (or higher if the

infrastructure is of considerable size), and after 1000 requests per second, the control plane will start building a queue in front of the placement logic. For example, Kubernetes would suffer queue build-up at even smaller request rates. Given the scope of this dissertation, we choose a simple design for the global scheduler to demonstrate the mechanisms to coordinate global and local domains. The global scheduler can become a bottleneck in a larger-scale scenario, demanding a more advanced design that allows scaling its performance. Well-known Cloud techniques [38] can be directly applied to address such challenges, as well as the use of federated architectures, both of which are discussed in section 10.2.1.

The other components in OneEdge can be scaled proportionally to the number of requests and applications by increasing the available control plane resources. Each local domain is independent of the others, and new μDCs can be added without further scalability issues, with the exception of the aforementioned implications pertaining to the global domain's scheduler. The monitoring subsystem can be scaled proportionally to the number of logical partitions (section 3.5.1), given that the only coordination point is the request's addition to the request queue and the corresponding scheduling operation. Under certain circumstances, the transaction manager could appear to be the bottleneck, but this would be due to the local domain scheduler (with whom it is coordinating) not keeping up with deploying/managing the application instances, which can be addressed similarly to the global scheduler limitation, or by the global domain's scheduler avoiding the overloaded μDC.

7.11 Chapter summary

In this chapter, we presented OneEdge, an agile hybrid control plane architecture for geo-distributed infrastructures that are likely to become the workhorses for the computational needs of emerging situation awareness applications. OneEdge can support all the requirements presented in section 4.2. It achieves this by enabling autonomous decision-making at the μDCs for standalone applications in tandem with a *global manager* that facilitates multi-μDC scheduling for coordinated applications. Furthermore, the global-domain manager's comprehensive view of the infrastructure allows for better decision-making related to load-balancing, better utilization, and agile application reconfiguration to meet E2E SLAs. A hierarchical monitoring component continually gathers statistics to detect load spikes and churn, triggering reconfigurations when application SLOs are likely to be violated. OneEdge's contributions include its novel distributed state management that allows concurrent scheduling decisions at μDCs's *local manager* and the *global manager* and the support for objective-based deployment for situation-awareness applications. We evaluated

OneEdge with both microbenchmarks and mock-ups of situation awareness applications in a multi-region Azure setup. OneEdge presents an overall solution for the programming model presented in chapter 3 and a complete implementation of the logical control plane architecture described in chapter 4.

The following chapters discuss the lessons learned from designing and evaluating multiple architectures and future directions and problems that can be built on top of our hybrid architecture.

CHAPTER 8
RELATED WORK

A complete control plane for managing applications on geo-distributed computational resources comprises several components. In this thesis, we focused on the architecture and distribution of control planes components, the mechanisms of coordination across those components, and the design of a programming model tailored for the setting. This chapter serves two purposes. The first two sections cover the state-of-the-art components this dissertation builds and improves on. Then, in the remaining sections, we discuss the existing literature on the other essential components for a control plane. This discussion helps understand how these components are currently being built and how they can be integrated into a complete end-to-end implementation of a control plane.

8.1 Programming models for situation-awareness

The complexity of managing distributed computing forced the creation of programming models to improve the developers' productivity. MapReduce [91] is the better-known example for simplifying the deployment of big data processing. Similar examples are data flows for stream processing [18, 19, 36, 92], Saphire [93] for mobile code offloading, and Serverless [94] for general function execution. They all hide the scheduling, management, and execution of the code from the developer, who only needs to worry about the application's logic.

These datacenter programming models are not designed for situation-awareness applications and do not provide the required SLAs. However, they provide similar abstractions in their domains that inspired the design of the programming model in this dissertation. The three most relevant aspects of these programming models are:

- Providing simple APIs to implement the applications.
- Hiding the complexities of managing the underlying resources and executing the applications.
- Describing the SLAs.

We followed similar design principles to MapReduce and stream processing interfaces. These datacenter's programming models give a simple API that effortlessly expresses the

developer's intent. For example, MapReduce [91] only requires implementing two application components: the map phase and the reduce phase, allowing the partitioning and processing of massive data with many distributed servers. Similarly, a *Beam pipeline* [92] (and also dataflow engines) defines a graph of all the data and computations in the application task, which can also use user-defined functions (UDFs). Given that stream processing is designed for continuous incoming data, similar to how situation-awareness works by continuously processing the client's contextual information, we used this dataflow design as the starting point for our programming model.

The APIs in the programming model allow hiding the complexity of application execution. For example, Spade [95] allows the developers to use fine-grain stream operators without worrying about the performance implications. Their compiler automatically selects the correct size of execution units (by merging) to minimize communication overhead when running distributed stream processing tasks. Similar intuitions exist for geo-distributed stream processing. For example, AWStream [96] proposes the use of adaptation as a first-class abstraction for the programming model, which hides the variations of network bandwidth from the developer. In an edge and IoT setting, FogFlow [97] simplifies the creation of automatic, elastic IoT services across cloud and edge, hiding the creation of instances across the infrastructure. Similarly, SpanEdge [98] allows separating applications into two sections, near the data source and far, hiding how these two types are deployed. In contrast to these works, the programming model proposed in this thesis hides the underlying geo-distributed infrastructure and provides first-class adaptation for E2E latency and spatial affinity reasons. The migration is handled under the covers, and the developer is only required to define the intent via the definition of SLA.

The last component of the programming model is how to define the SLAs. In a cloud setting, SLAaaS [99] describes a programming model that explicitly exposes SLAs programmatically, including performance, dependability, and energy. The definition of the cloud SLAs offloads their management from the developer to the runtime. Similarly, one design for SLA management similar to our programming model is Henge [100]. In Henge, the developer specifies its intent for each stream processing as an SLO requirement, mainly in latency or throughput needs. The runtime of Henge then adapts to continuously meet each stream task's respective SLOs across all applications running. Our design follows the same principle of SLAaaS and Henge but extends them to a geo-distributed setting to support both latency and spatial-affinity requirements.

8.2 Control plane architectures and mechanisms

As previously discussed in section 4.4, cloud control planes in all their architectures (*i.e.*, monolithic, partitioned, and shared state) are not appropriate for geo-distributed computational resources and situation-awareness applications either because they are not scalable with heterogeneous connectivity or they do not support the requirements of the applications (section 4.2). This section extends the analysis to other control planes architectures and how they are structured to handle deployment requests (including the request interfaces).

There has been a growing interest in developing container managers for geo-distributed and edge computational resources beyond KubeEdge and KubeFed. For example, StarlingX [101] is an open-source project similar to KubeEdge that supports distributed edge clouds. StarlingX has the same limitations as KubeEdge of relying on all decisions going to a central controller. Other similar research projects are Fogernetes [102], mck8s [103], and ge-kube [104]. Fogernetes incorporates support for geographical information when deploying applications but only does it in a coarse-grained manner with labels and no support for mobility. Similarly, mck8s tries to reduce resource fragmentation across geo-distributed datacenters. The control plane of both Fogernetes and mck8s have a centralized architecture. On the other hand, ge-kube tries to offload part of the central control plane to the geo-distributed μDCs in a leader-follower architecture. The follower carries out the decisions of the centralized controller. However, decisions are still taken in one centralized location. Ge-kube supports latency and network topology-aware placements but does not support multi-components applications, application state management, and mobility monitoring, making it unsuitable for situation-awareness applications.

Additional research in control plane design has been done in more specific programming models, like stream processing and serverless. For example, Nastic et al. [35] present an architecture for a serverless platform for the edge-cloud continuum. The application architecture could support managing situation-awareness applications, but they do not provide an implementation of the architecture or for the associated mechanisms. In the context of geo-distributed stream processing, EDF [105] proposes a control plane with a two-level architecture, with a control plane executed for each application instance. The hierarchical solution has a centralized leader that runs in one of the application components. It coordinates the global run-time adaptation of follower application components. Each follower, associated with a given application component, runs a local control loop to adapt the component to any changes using extendable policies. This design is similar to the global and local domains in OneEdge and provides implementations of the local policies that could be

incorporated into the local domains of the μDCs. EDF supports migration but does not handle violations due to mobility-induced latency or spatial affinity. Additionally, it follows a pause-and-resume approach for the state migration, unsuitable for situation-awareness applications given that the application needs to stop the processing until the full state is migrated.

8.3 Scheduling algorithms

The scheduling algorithm defines the placement of application components into the available computational resources. Traditionally, in a cloud setting, the scheduling problem is formally modeled as an optimization problem where the final solution maps a task or application component to a specific resource. The scheduler tries to optimize for a specific metric, for example, fair-sharing [106]. More recent work supports multiple simultaneous objectives [107] and multiple types of resources [108, 109].

Scheduling applications on geo-distributed resources adds complexity to the problem because the resources are disaggregated, and awareness of the network topology is required [110]. The metric of interest in the initial frameworks tackling this problem was the bandwidth between datacenters [111] and the corresponding data [112] and tasks placement to avoid traversing the WAN. Situation-awareness applications and edge computing, in turn, add additional complexities in terms of latency [113] and mobility [114].

Edge computing requires densely geo-distributed datacenters. The support of QoS in this domain requires the scheduler to be aware of both client's mobility and the network topology [115]. One SLA that is critical for situation-awareness applications is latency. Previous work like FOGPLAN [116] and Cardellini et al. [117] formulated the optimization problem to support QoS-aware dynamic edge provisioning and described efficient solutions for the problem. Additionally, to support mobility, systems like Folo [118] predict the clients' mobility (*i.e.*, cars in Folo) to assign them to resources while considering latency and resources constraints. Both mechanisms can easily be incorporated into the hybrid control plane architecture presented before as part of the scheduler placement logic. Unfortunately, spatial affinity is not tackled by state-of-the-art, but can be easily defined as new constraints in the optimization problem.

Another aspect of μDCs (*i.e.*, edge computing) that affects the design of the scheduling algorithm is the resource scarcity at each μDC. The scarcity requires additional optimizations to improve resource utilization. One such optimization is partitioning the application and only running relevant components near the clients and the non-latency sensitive in the

cloud [119, 120]. For example, SpanEdge [98] allows defining which computation has to be near the data sources and implements a scheduler to support it. Additionally, ideas like Synergy [121] are helpful for edge computing by reusing components and computations across multiple clients. Another relevant area is selecting the right parameters and amount of resources when starting an application. For example, VideoEdge [77] selects the right configuration for the application component and merges similar components to reduce overhead in the computational nodes. All the optimizations can be included in the global domain's placement logic of OneEdge to decide how and where the applications are deployed.

Finally, the geo-distribution of the resources also affects the scheduler's architecture. Similar to OneEdge [22], there have been algorithm designs for schedulers that are decentralized or hierarchical. For example, Cardellini et al. [122] propose to have a global manager for the overall application decisions and per application component local managers and show that systems can be stable when making hierarchical control decisions. Similarly, a new wave of schedulers is being designed for serverless running in edge infrastructure [123, 124, 125]. For example, Skippy [125] proposes a scheduler with additional constraints to match the application's data flow and the network topology, as well as presenting ways to tune the schedulers. Both types of optimizations, hierarchical algorithms and local-domain serverless, can be used to tailor the behavior of OneEdge towards applications that are not necessarily situation-awareness applications but can leverage all the other components.

8.4 Monitoring

Adapting application components to maintain their SLAs involves four stages as part of the feedback loop: monitor, analyze, plan, and execute (MAPE) [126]. The latter two are part of the scheduler and architecture. This section focuses on the first stage, monitoring, of self-adapting control planes.

Many monitoring frameworks have been designed for cloud environments [127, 128, 129, 130]. However, they focus on metrics collection and aggregation at the granularity of an application component, which do not cover all the needs of situation-awareness applications. The main takeaways from the design of such systems are the techniques for efficient measurement and aggregation of metrics. For example, Andreolini et al. [131] dynamically balance the amount and quality of the monitored metrics by sampling the generated times series, which reduces the monitoring cost. Similarly, JCatascopia [132] allows modifying

the data granularity and aggregation periods via adaptive filtering. Both techniques help reduce the measurement overhead on the application that is being measured. As mentioned previously, OneEdge aggregates time windows into percentiles used to detect failures. The presented cloud techniques complement our contributions and can be used to further improve monitoring efficiency.

In the context of edge computing, monitoring needs to deal with two additional issues, measuring E2E latency and aggregating results across geo-distributed components. For measuring E2E latency, the theory behind *critical path* calculation can help its understanding. The critical path is the longest path in the application's dataflow graph and represents the sequence of the application component executions that take the longest time to complete [133]. Most recently, SnailTrail [78] generalized the online critical path analysis for long-running and streaming computations providing immediate insights into the components that are becoming latency bottlenecks. Thus, algorithms like SnailTrail can improve the detection quality of E2E latency violations, as well as work as Sonata [134], which allows correlating outliers with the nodes that are affecting the latency. The concept of *critical path* has also been used in stream processing engines like Borealis [135]. We used a simplified version of the critical path analysis in OneEdge.

To address the geo-distributed monitoring of applications, edge monitoring frameworks have been designed with both decentralized and distributed architectures. For example, FogMon [136] creates a peer-to-peer (P2P) network out of the computational nodes being monitored, with an additional hierarchy of leaders and followers. Only the leaders disseminate information through the P2P network. FogMon can then reduce the overall overhead and improve the framework's scalability while also providing E2E latency monitoring. Other frameworks, like ADMin [137], reduce the monitoring overhead in the scarce resources available in the µDCs for both monitoring and aggregation. OneEdge uses a leader-follower design for monitoring E2E latency metrics. The designs of FogMon and ADMin could be incorporated to reduce further the overhead of monitoring individual servers and for additional aggregation before reaching the global domain.

8.5 Dynamic reconfigurations

This section focuses on the second stage of MAPE, namely, analyzing. The analysis phase is the one that defines if a reconfiguration is required. Given the similarity between situation-awareness and stream processing, we start analyzing reconfiguration in this domain. For stream processing applications, reconfiguration is common, given that

they run continuously [138]. Stream processing systems can adapt running applications due to changes in various metrics, like utilization [139] and latency [140]. For example, SPADE [139] adapts the number of resources assigned to a specific application component based on the request arrival and the available spare capacity. This reconfiguration policy is similar to the dynamic reconfiguration in OneEdge's local domain. Similarly, Loharmann et al. [140] present mechanisms for reactively enforcing latency guarantees in data flows and measuring the metrics needed to detect the need for scaling actions. This type of action is similar to the ones taken by OneEdge after a violation is found. In general, research in autonomous reconfiguration in distributed stream processing [141, 142, 143, 144, 145] can be used to modify the pluggable monitoring policies in OneEdge's monitoring pipeline to detect the need for reconfiguration better.

The second operation that the *analysis* phase needs to perform is to decide when to perform the reconfigurations, which involves predicting the future configurations of the clients and infrastructure. There is prior work in selecting *when* to migrate an application component (or the entire application) due to client mobility. Urgaonkar et al. [146] model the migration problem as a Markov Decision Problem (MDP). They decouple the initial MDP into two independent MDPs that allow the problem to be solved using a Lyapunov optimization. Wang et al. [147] propose further refinements to this solution by developing a polynomial-time algorithm with some relaxation in the system assumptions regarding the error bounds on the costs of hosting and migration. Similarly, MCEP [148] includes both the design and the mechanisms to automatically adapt the processing of events according to a consumer's location to reduce latency, network utilization, and processing overhead by providing on-demand and opportunistic adaptation. In the approach presented in this dissertation, any of these strategies could be used for deciding on *when* to migrate.

In general, we can design *analyze* phase with a control-theoretic approach, like the one presented in SLAaaS [99] or AWStream [96]. SLAaaS continuously finds the service configuration that provides the highest utility in a changing environment. That configuration is maintained and adapted by the control algorithm. Then, the monitoring subsystem can define how to act by predicting the expected performance in a given configuration, like DS2 [149], and when to take actions like Gu et al. [150]. By integrating the mathematical (or machine learning) models of the environment, the monitoring subsystem can react proactively and in a timely manner to constant changes in the application's context.

8.6 Application migration

There are three common mechanisms for migrating application components between different geo-distributed computational resources: VM, container, and state migration. A VM migration involves transferring the whole VM state between different locations. For example, *VM handoff* [151] shows mechanisms to perform this big data transfer on unreliable connections between edge locations (or cloudlets in the *VM handoff* work). The main limitation with VM is that the state involves the whole memory state of the operating system. Recently, containers have been chosen as the default deployment orchestration unit. Multiple papers have addressed the container migration problem in geo-distributed and edge computational resources [152, 153, 154]. The main focus of those papers is to reduce the overall size of the transfer data and the downtime during the migration. For example, MyceDrive [153] reduces the downtime by avoiding killing the previous container until the migration is complete while being aware of the network bandwidth between sites. Similarly, Ma et al. [154] reduce the size of the images transferred by leveraging the Union filesystem used by container images. The main benefit of migrating full VMs and containers is that they can treat the computation as a black box, and the migration is transparent to the application components. However, this also increases the cost of migration, exacerbated by the heterogeneous network in a geo-distributed infrastructure.

In this dissertation, we have a deeper knowledge of the workload, and through the programming model, we avoid the need for snapshots of the full underlying runtime and execution, which reduces the amount of transferred data by an order of magnitude. This idea has also been applied in CEP [155] and stream processing [156], where the runtime system knows what state is relevant to be migrated and can be proactively migrated to its new location. There is no need to migrate the runtime state, as it can be easily recreated without coordination. For example, Castro Fernandez et al. [157] proposed a similar idea to state migration in Foglets, where they expose the internal application component state explicitly to the runtime through a state management API, which the runtime can then use only to migrate the relevant state to the new instance. The main difference between Foglets and Castro et al.'s approach is that the latter focus on a datacenter environment, so the copy of the state is simplified. On the other hand, Foglets has to consider the heterogeneous network and the possibility of a new migration happening before the previous state migration is completed.

CHAPTER 9
DISCUSSION AND LESSONS LEARNED

This thesis presented both a programming model and a control plane architecture to serve situation-awareness applications running on a geo-distributed infrastructure. This chapter examines how these two factors affected our design decisions. Then, we discuss insights from our work that have broader applicability to the systems research community, particularly for real-time geo-distributed systems, and how we should leverage application semantics and infrastructure knowledge to improve systems performance.

9.1 Control plane design for situation awareness application

The programming model is the interface that allows the developer to interact with the control plane, affecting both the control plane and the developer.

From the perspective of the control plane. The programming model is the key component that provides *intent* to the control plane from the developer. This *intent* provides flexibility to the control plane and allows it to take different decisions depending on the context in which the application is to be deployed or managed. By their inherent nature, situation-awareness applications continuously interact with the physical world, and the context in the physical world can alter how the applications should be handled; it can affect how many instances to deploy or when to deploy the applications. For example, chapter 6 showed how an incremental approach could reduce resources usage if some application components in the DFG are not commonly used. The incremental approach defines both when and how to deploy the application components. Similarly, for coordinated applications in chapter 7, the client's location will define which logical partition will process it, which then defines which physical instance should be in charge—changing with the client's mobility. The programming model gives implicit and explicit semantic knowledge from the application to the control plane to better fulfill its requirements.

Situation-awareness has novel problems for distributed control plane design in that it covers a different set of metrics that may not be common in throughput-oriented applications. For example, in throughput-oriented applications, the control plane provides guarantees on a certain level of maintained throughput, availability, and latency (calculated only inside the datacenter) for a high percentile of the requests (*i.e.*, 99.9%). On the other hand,

for situation-awareness applications, it goes further and involves monitoring metrics outside of the datacenter (or µDC) where the applications are hosted. For example, the control plane needs to monitor the communication between the client and the datacenter to calculate the E2E latency of the application. Similarly, the control plane needs to continuously monitor the client's location to provide the spatial-affinity requirements. Future generations of situation-awareness applications would keep pushing towards a control plane that can monitor more physical metrics to better serve the application needs and its clients.

From the developer's perspective. The level of abstraction provided by the programming model defines what aspects of the infrastructure and deployment process the developer needs to worry about and what they can express as an application (*i.e.*, how flexible it is to represent the applications). For example, in this dissertation context, the developer does not need to worry about the infrastructure geo-distribution, as this is handled under the covers by the application requirements. However, the developer still needs to provide the explicit topology of the DFG and the code implementation of each of the application components. We were able to hide this complexity by using well-crafted APIs and event handlers.

Similarly, the programming model also hides the creation and definition of the number of instances required in the framework. This abstraction is done by creating an application taxonomy and defining functions to map clients to logical identifiers (*e.g.*, AoIs). It hides the discovery from the application (and inherently from the developer) and any migration decisions that happen due to changes in the QoS. As a result of this flexibility, the control plane automatically selects an appropriate geo-distributed µDC to host an application component and potentially reuse already running application components for multiple clients (thanks to the logical partitions presented in section 3.5.1).

In general, the DFG programming model abstraction extended to support situation-awareness application requirements was shown to be flexible enough to host the applications we were considering and provide enough semantic information for the control plane to leverage the application's expected behaviors to improve the efficiency of the designed mechanisms.

9.2 Control design for geo-distributed infrastructure

A densely geo-distributed infrastructure has a different set of constraints to consider when designing a control plane:

1. There is weaker and heterogeneous connectivity between infrastructure components

and transitively between control plane components.

2. There is a higher likelihood of a reduced number of resources at a given µDCs (*i.e.*, scarcity).

3. The mobile clients have changing contexts that affect their perceived QoS.

One key aspect that separates geo-distributed settings from regular cloud datacenter environments is the need for a local-first design. Local operations ameliorate the impact of item 1 presented before. If the µDCs can handle all the required operations and serve clients locally, then transient issues on the network or heterogeneous latencies should only occur in counted operation types and not be in the critical path for most. The main limitation of this approach is that the other two items, 2 and 3, may require coordination across locations, given that, as previously discussed, certain applications may need to be deployed across multiple µDCs or need migration from one µDC to another. A fully local design can be myopic with decisions, and the scarce resources make this limitation more noticeable. This limitation causes that the main driver for a geo-distributed setting is supporting autonomous decision-making (requirement **R1**) while allowing global knowledge for coordination. Then, a design for a geo-distributed setting needs to support both autonomous decision-making (requirement **R1**) and global knowledge for coordination (requirement **R2**).

A hybrid multi-domain design fulfills the seemingly opposing needs of autonomous decision-making and global knowledge for coordination. OneEdge, our proposed hybrid control plane, supports these needs thanks to its distributed monitoring component and local-first state management mechanisms. The continuous monitoring provides better visibility of both dynamic conditions of the available resources at the µDCs and clients' mobility and allows the control plane to be agile and responsive to the changes of item 3. Similarly, both decentralized mechanisms (*e.g.*, deflection) and centralized mechanisms (*i.e.*, periodic load balancing) allow better management of scarce resources, addressing the constraint in item 3. The local-first state management moves the authoritative replica of the data to be local to each µDC. To support cross-site coordination, we modified 2PC to be optimal in the number of roundtrips (*i.e.*, one RTT) by simplifying the algorithm and leveraging the semantics of resource allocation and the programming model structure.

Next, we present the insights that guided the design of such mechanisms for both local and global domain components.

9.3 Lessons learned

9.3.1 Leverage application semantics and infrastructure knowledge

One key aspect when designing control planes for managing resources is the insights that can be extracted from the application's expected behavior. In this dissertation, the control plane leveraged application information in objective-based requirements and geospatial mobility information. Additionally, understanding the scheduler intention enabled the different 2PC enhancements in section 7.7, achieving a tangible improvement in the latency of WAN-related operations. Similarly, the exposure of application logical partitions section 3.5.1) allowed efficient load balancing and migration that would not be possible without application information.

Another important insight from situation awareness applications is that close-by resources are the most useful for its clients. For a standalone application, this knowledge allowed the use of an efficient local domain. The local domain supports autonomous decisions by knowing that the common case scenario will be the client being deployed in a close-by µDC while still supporting operations to handle edge cases like a µDC having no capacity with deflection.

In general, the intuition is that the interface should expose enough semantics of the application behavior to the entity managing the resources—ideally in an implicit manner from the developer's perspective. This intuition is already in use on compilers which can leverage the programming language specification for its benefit when the underlying hardware changes. When developing frameworks and control planes, we should understand the application domain such that we can squeeze all such potential performance gains. Additionally, the control plane should understand the expected behavior of applications and design the mechanisms to be biased towards the common case scenario.

The other important venue to consider when managing computational resources is infrastructure knowledge, as it is needed by the control plane to efficiently manage all the computational resources. For example, the lack of geospatial information negatively impacts the management of situation-awareness applications, as it was highlighted in chapter 5 and the evaluations in section 7.10.3. In a datacenter environment, infrastructure knowledge is leveraged when rack and network topology are considered when performing placement decisions. However, future infrastructure evolutions will expand the metrics spectrum even further for the control plane. For example, the network topology and the available connectivity will become even more heterogeneous. For example, protocols and

technologies used through the network will vary across the infrastructure compared to current homogeneous datacenter topologies. The first mile could be 5G wireless connectivity, while deeper in the network will be high-bandwidth fiber optics. Additionally, this heterogeneity will only grow in general as new application-specific hardware (*i.e.*, ASICs) and a plethora of accelerators start becoming the norm [158]. Therefore, taking advantage of infrastructure topology and composition information will only grow more important with future hardware evolutions.

In conclusion, it is the role of the control plane designer to consider each vector that can improve efficiency from both the application and the infrastructure being handled.

9.3.2 Focus on the real objective

Well-known algorithms are designed for a general use case. Therefore, the algorithms need to be correct under conservative assumptions as the designers did not fully know the context in which the algorithms will be used. However, once an algorithm is chosen as a building block in a specific context, there are usually possible venues for improving its performance by noticing assumptions that can be relaxed in the original design.

For example, the generality of 2PC is not required for the coordination mechanisms presented in chapter 5. More concretely, the control plane allocation coordination does not need to agree on the same final output as in the original 2PC design, as it only needs to reach the same final configuration. In this dissertation, the required agreement is not in reaching the same resource allocation and configuration, but it is the allocation of the required resources to cater to all the clients' SLOs that are currently trying to use the framework.

In general, we need to find the parts of the known algorithms that can be relaxed to improve the overall performance in a given context. Another example in this dissertation is the decision of what operations are to be executed in each of the phases of the 2PC. For example, a database cannot directly modify the value during the first phase because it will become visible to other entities in the system, violating isolation constraints. In our context, the deployment of an application is not visible to other system components, which allows us to bypass the second phase from the client's perspective, and considerably reduces the overall latency cost. These examples give an idea of how well-known algorithms should be tailored to the context where they are deployed and benefit from that semantic knowledge, and open up the space for new research on configurable coordination algorithms that can be tuned to each specific application infrastructure and need.

CHAPTER 10
CONCLUSION AND FUTURE DIRECTIONS

This work presented a programming model and progressively built a hybrid control plane architecture for geo-distributed resources and situation-awareness applications. This chapter first summarizes the main contributions of this dissertation and then presents future directions for research in this domain.

10.1 Conclusion

Situation-awareness applications will continue to blend seamlessly with our daily lives. This trend will keep growing with newer technology like Internet-of-Materials [159] that will embed passive sensors in all our surroundings. However, for these sensors to be useful, computational resources need to be close to them, such that they can be accessed with low latency and high bandwidth.

The computational resources will need to be densely geo-distributed worldwide to be close to the sensors and users that need them. However, there is a gap in how state-of-the-art control planes manage those resources and the actual needs of these novel applications. First, they lack the interfaces and metrics required by QoS-sensitive control policies for situation-awareness applications (*e.g.*, spatial and E2E latency). Second, their centralized design has an *impedance mismatch* with how the control plane components and the computational resources will be distributed across the world. This dissertation methodically analyzes the domain and proposes a new programming model and control plane architectures tailored for this new context of situation-awareness applications and densely geo-distributed computational resources to address this gap.

The first contribution of this dissertation is in chapter 3. We propose that for a programming model to be useful in this context, it has to include geospatial requirements from the developer, as well as provide flexibility to the control plane to partition and replicate the components across the geo-distributed infrastructure. First, chapter 3 presents a taxonomy of situation-awareness applications with two types of applications: *standalone* and *coordinated*. *Standalone* applications focus on processing single clients with tight-latency control loops that affect the client environment and decisions, and *coordinated* applications focus on applications that aggregate information from multiple geographically close clients

to enhance the overall decision-making of each client. Then, we presented an extension of the DFG model. By its inherent nature, DFG allows the partitioning of the application into each of its nodes. Additionally, we define semantic ways —based on a deep understanding of the application behavior— to partition the application to improve this flexibility. Finally, we add spatio-temporal requirements with an intuitive interface to cater to the application needs. Because of these improvements, the programming model helps both the developer and the control plane to better implement and manage situation-awareness applications, as shown in chapter 3.

Chapter 4 describes our second contribution, an in-depth analysis of logical components needed to support the requirements of a control plane for geo-distributed resources and situation-awareness applications (section 4.2). It builds on both the application learnings from chapter 3 and the properties of the computational infrastructure. We analyzed how these components interact with each other and with the geo-distributed infrastructure. The logical components facilitated the understanding of implementations and distributions of components in the infrastructure, which allowed us to design a proper control plane for this dissertation domain.

The final contribution is distributed through chapters 5 to 7, where we progressively design an architecture for handling geo-distributed resources for situation-awareness applications. The design started from a state-of-the-art, fully centralized architecture in chapter 5, where we showed quantitatively and qualitatively the unsuitability of such a design for geo-distributed situation-awareness applications.

Based on our learnings from both the centralized architecture and the proposed programming model design, we considered the other end of the spectrum with a fully decentralized architecture in chapter 6. The focus of the decentralized architecture was to tackle the lack of support for autonomous decisions in state-of-the-art centralized architectures and to handle the temporal requirements of situation-awareness applications. Chapter 6 showed that splitting the control plane into several local domains matches the inherent geographical structure of situation-awareness applications. Additionally, we highlighted and evaluated how continuous E2E monitoring of DFGs is an essential task that needs to be performed by the control plane. This monitoring layer is used as a building block to design the mechanisms required for application adaptation concerning the users' continuously changing context and how adjacent local domains can interact to improve the quality of the service for the clients. We also showed how we could leverage the programming model to reduce the latency to access application instances by 93%.

However, a decentralized architecture can not provide the spatial requirements of grouping clients based on geographical proximity (*i.e.*, spatial affinity in chapter 3). Therefore, this dissertation finally proposes a hybrid control plane architecture that merges the design benefits of both centralized and decentralized architectures. The hybrid control plane architecture combines autonomous decision-making at each μDC to reduce standalone applications' deployment latency with global decision-making for scheduling coordinated applications. It achieves this by reducing the impact of performing geo-distributed coordination and keeping the authoritative state locally at each μDC. The proposed concurrency control algorithm uses an enhanced 2PC protocol that leverages application semantics and infrastructure knowledge to reduce the common-case response time to one RTT instead of the usual two RTT required by a baseline 2PC for scheduling decisions (that need global knowledge). Additionally, we extend the decentralized architecture's monitoring layer to monitor spatial metrics to adapt to all the situation-awareness functional requirements.

Finally, using a mix of applications on multi-region Azure instances representing a realistic setting, we show that a hybrid control plane architecture can fulfill the unique requirements of situation awareness applications in contrast to centralized or fully decentralized control planes. For example, compared to a centralized architecture, a hybrid architecture reduces deployment latency by 66% for single-μDC standalone applications without compromising spatial and temporal SLOs. More specifically, we showed that it can closely match the spatial-affinity requirement for coordinated applications, and perform low-latency operations for standalone applications without WAN operations, which validates the thesis statement that we can build an efficient control plane for this context by combining components of both types of architectures, and they can work efficiently together.

In conclusion, this dissertation defined an architecture and associated mechanisms to manage situation-awareness applications in a densely geo-distributed infrastructure. Furthermore, it defined the base building blocks for creating an efficient control plane with a local-first architecture that can support the spatio-temporal needs of situation-awareness applications.

10.2 Future Directions

This section discusses future directions for the two main aspects of control plane design covered in this dissertation: geo-distributed resources and situation-awareness applications.

10.2.1 Control plane design for geo-distributed resources

Partitioning the global domain

In this dissertation, we designed the enhanced 2PC mechanism to allow efficient concurrent transaction execution in different localities (*i.e.*, global and local domains) for resource allocation decisions. An interesting future direction is applying the same mechanism to improve further the scalability and responsiveness of other control plane components. Next, we propose two such areas.

Federating the control plane. Given the inherent geographical locality of μDCs, we can partition the *global manager* component (or add an extra domain in the hybrid architecture) into multiple sub-managers with overlapping coverage regions. The system would require overlapping areas in the boundaries of the regions to support cross-region migrations. Overall coordination between cell managers and the *local manager* could be implemented using a similar mechanism to the enhanced 2PC, where multiple managers concurrently execute their operations on the required μDCs.

This federated design allows the cell manager to be closer to the handled μDCs while still providing the benefits of global knowledge. The corresponding cell manager could be collocated with one of the *local managers* in a given cell. However, it incurs more complexity in the handover of monitoring and policy and raises the question of how the new cell manager knows it needs to manage a specific client when migration happens to the overlapping area. Additionally, such a design could potentially increase the likelihood of a transaction failing and potentially decrease the efficiency of resource assignment and utilization. Further research on improving the coordination across the federated managers without impacting the overall control plane's efficiency is an interesting problem to tackle. Even just partially moving the global controller operations to a select *local manager* could be viable for a subset of operations, such as managing intra-cell deflection.

A federated global architecture improves the fault tolerance and reliability of the global control plane. For example, the blast radius for a sub-manager failure would now be a metropolitan area instead of all the μDCs. Additionally, it does not need explicit primary-secondary replicas, given the support for overlapping regions. Instead, multiple sub-managers can be in charge of a given metropolitan area, with a load balancing mechanism in the μDCs to choose one of the global sub-manager replicas. Each sub-manager instance would be independent and in charge of monitoring the clients deployed by it.

On the other hand, each μDCs would detect failures of sub-managers and migrate the

monitoring and processing of the corresponding clients to a different instance. This design adds additional operational costs to the μDCs but can considerably improve the scalability and reliability of the overall control plane. The global manager handles the failures of the local domain managers, and in a decentralized fashion, the local domain managers handle the failure of a sub-manager. This fault-tolerant federated architecture will need to be tested to verify that each domain can correctly detect failures and reduce the time these detections and migrations take.

Scheduler parallelism. The scheduler of the hybrid architecture was designed such that the implementation of the placement logic had a simple interface (*i.e.*, it has the illusion of sequential execution of successive requests), and it did not have to worry about the changes to the underlying aggregate state. We could use *enhanced 2PC* to maintain this illusion while increasing the scheduler's scalability. The scheduler's interaction with the aggregate state is no different from the *local managers* and the aggregate state. We can have multiple concurrent instances of the scheduler. All of them modify a local copy of the aggregate state and then try to modify the "global" copy of the aggregate state. An additional validation component would be required between the schedulers and the transaction manager to verify that the allocation operations do not create an oversubscription in a μDC and apply the changes to the "global" aggregate state if it passes the verification. The validation component would be the receiving end of the *enhanced 2PC* (*i.e.*, the same end of the 2PC as the *local manager* in its original use). We did some preliminary results and scaled it to up to four parallel instances without noticeable loss in aggregate throughput, similar to the results in Omega [38]. However, further evaluation is required to understand the complete behavior when running this design for big-scale E2E deployments.

Generalizing to more metrics

The hybrid architecture was tailored for three main types of metrics: computational resources (*i.e.*, allocation, utilization, bandwidth), latency, and geospatial data (for computational resources and client). OneEdge's architecture can monitor these metrics across the geo-distributed infrastructure and reason about their E2E aggregation and processing. However, suppose the geo-distributed infrastructure changes in the future, and it needs to handle a new set of dynamic metrics. In that case, the control plane should easily allow adding more metrics to the monitoring pipeline, with their corresponding implementations for aggregation and efficient dissemination across the geo-distributed infrastructure. For example, if the weather of an AoI becomes a metric of interest, and if changes in the

weather need to trigger reconfiguration of the DFGs (*e.g.*, changing the machine learning model due to rain), then weather information should be easily be added to the monitoring component, and the control plane should expose the new metrics to both the scheduler and the entities applying the policies. This new requirement raises the question of what are the right interfaces and mechanisms for querying, aggregating, and publishing the metrics, such that we can easily extend all the control plane components without having to re-engineer the plumbing of information, coordination, and execution of applications and that it becomes immediately available to the decision-making components.

Extending to different infrastructure models

The current view of geo-distributed edge infrastructure, such as Azure public multi-access edge [3], assumes that you will use the same platform to deploy all your application components. However, similarly to how the paradigm of multi-cloud evolved to avoid lock-in, multi-edge could emerge for similar reasons or to have the best latency access for each location in the country. OneEdge design assumes that it has full visibility to all the resources available in the infrastructure—including their geographical location— and that it can directly manage them, similar to a single geo-distribute edge design. A multi-owner infrastructure (*i.e.*, from multiple third parties) adds an extra layer of complexity, as there is an additional step on requesting a VM from the providers. Once the VM is deployed, the algorithm is similar to the one presented in the dissertation. The problem is exacerbated if the interface to the provider is not Infrastructure-as-a-Service, but more ephemeral compute access like Function-as-a-Service.

Further research is required to understand what interfaces should the third-party owner provide such that a high-level control plane with application knowledge can efficiently perform its tasks and how a hybrid architecture would fit with that schema. An orthogonal but similarly important problem is knowing when to request additional VMs, when to turn them off, and how to compare resources from independent providers. This last problem is similar to decisions on how long to keep serverless containers alive [75].

Improving local domain state management efficiency

OneEdge's reliability mechanisms rely on a primary-secondary replication of the infrastructure state for the local domain managers. Given that there are many geo-distributed *local managers*, it is inefficient not to consolidate this replication effort across multiple instances instead of having independent replica groups. This limitation can be solved by

externalizing the control plane state and having stateless replicas in charge of executing the tasks. This design also allows for a diversity in replication across geographical regions to further improve reliability against correlated failures. Geo-distributed infrastructure opens a new possibility to how we replicate control plane components' instances for reliability.

Replication needs to incorporate knowledge about the network topology to reduce correlated failures, and the cloud policies like rack-awareness [160] are not enough when network connectivity is not homogeneous. Our previous work in Metric [161] presented mechanisms to more efficiently replicate data in a geo-distributed infrastructure. However, additional work needs to be done to understand the number and location of replicas and the policies to define when to replicate and how the data is stored. Furthermore, the use of stateless agents on top of a reliable data layer also raises questions about allowing *local managers* to be remote (*i.e.*, in a different μDC) temporally while the manager in the μDC is being fixed. Similarly, another question is, if the μDC becomes unavailable, should the other locations proactively take ownership of the application components running in those locations, or should they wait for the *local manager* to become healthy again, given that there is no entity monitoring QoS.

10.2.2 Control plane design for situation-awareness applications

Enhancing the programming model

The programming model presented in this thesis caters to the need of situation-awareness applications running on a geo-distributed infrastructure. However, for many developers, the definition of a DFG may be too low-level for their tasks, as the developer needs to know how to partition the application into nodes of a DFG. Additionally, they still need to implement boilerplate logic for serializing messages and sending them between the different components, as well as defining the function matching the geographical location of users to AoIs for coordinate application. Therefore, a higher-level declarative language may be required to facilitate the implementation of applications, similar to how SQL emerged for data processing, CQL-like queries for continuous queries [162], and more recent work is trying to do the same for video analytics [163].

We could build a declarative language on top of the DFG programming model presented in this dissertation. However, a declarative language in this context has new complexities involving how to expose geographical data to create multiple partitions of clients easily and how to map the E2E latency in the DFG to the high-level declarative interface. Then, finding the right abstraction to group clients and expose requirements becomes a research

problem that would push geo-distributed situation-awareness applications to broader adoption. Additionally, it opens a new door to more easily composing applications. For example, each declarative query output can be exposed and reused as input to newer queries, facilitating the creation of more complex applications without the difficulties of manually handling a DFG. Similarly, it also facilitates the aggregation of multiple diverse sensor types that automatically join thanks to the runtime library and are then exposed as virtual sensors that can be further aggregated.

Expose more semantic information to the control plane

The control plane design in this dissertation assumes that each application component is a black box. This assumption allows for a general framework that can cater to multiple types of applications. The drawback is that the control plane cannot control how the application executes when under resource scarcity. One interesting venue in this context is to allow the control plane to be in charge of load shedding. Load shedding can be performed when the queues in front of the application components grow large enough that it is no longer possible to process those messages before the E2E latency requirement. Inherently, load shedding can be performed oblivious to the content of the requests, but this is inefficient in the context of situation-awareness applications. Situation-awareness applications process data that mostly comes from sensors that measure activities in the physical world. Sensors tend to have a high degree of redundancy on consecutive measured data. For example, a camera generates 24 frames per second, and two consecutive frames are quite similar and may contain the same objects, given that objects in the physical world do not move that fast compared to the time distance between consecutive frames.

The control plane could leverage the redundancy of sensor inputs to define priorities on the input data (*e.g.*, frames) to select the right set of pending to process inputs to discard without heavily impacting the output. To implement this, the developer of the application could provide a function that scores the input, and then the control plane can use that to maintain a control loop that keeps the latency bounded while not affecting the application accuracy as much. We had done some preliminary work on this topic for real-time video stream processing. The semantic load-shedding mechanisms allow a much better trade-off than random sampling, even with cheap non-machine learning scoring. However, further research is still needed to generalize the approach and find the right interface and mechanisms required for the control plane not to use excessive resources to execute the load shedding scoring and dropping. We want most of the available computational resources to

be devoted to the actual execution of the application components and not to pre-processing performed by the control plane. A toolkit for building such scoring functions with bounded resources and low impact on accuracy could provide additional headroom to the scarce resource at μDC and another knob for the control plane to prioritize certain applications over others depending on the scores. This comparison opens another research direction in finding how to make this scoring function comparable across domains.

Appendices

APPENDIX A
PSEUDOCODE FOR CONNECTED CARS APPLICATION

This section includes all the associated pseudocode for the connected car application explained in section 3.6.

```
{
  "location": "/usr/local/lib/connected_cars.so",
  "edges": [
    ["dbscan", "prune", "objects"],
    ["prune","concat","objects"]
  ],
  "start_node": "dbscan",
  "edges_to_client": [("prune","objects")]
  "requirements":
  {
    "latency": [
        [["prune","client","objects"],100]
    ],
    "spatial_affinity":[
        ["dbscan","subregion"],
        ["prune","subregion"],
        ["concat","region"]
    ]
  }
}
```

Figure A.1: Connected cars—Json application configuration.

```
1 void bootstrap(string name)
2 {
3     ApplicationComponent componentInstance;
4     // Create the corresponding application component to be run
5     if(name == "dbscan")
6     {
7         componentInstance = DbscanNode();
8     }
9     else if(name == "prune")
10    {
11        componentInstance = PruneNode();
12    }
13    else if(name == "concat")
14    {
15        componentInstance = ConcatNode();
16    }
17    // Register the component to be the one to handle all the messages
      in this instance
18    registerComponent(componentInstance);
19 }
```

Figure A.2: Connected cars—Bootstrap function.

```
1 class PruneNode : ApplicationComponent
2 {
3   // Object should previously be initialize in the constructuctor
4
5   // Handler for receiving messages from the dbscan node
6   void on_send_up(msg m, partionId pId)
7   {
8     // Deserialize message
9     input = m.deserialize();
10    // Apply the pruning algorithm
11    prunedList = pruneListOfObjects(input.getObjectList());
12    // Serialized pruned list
13    serializedPrunedL = serialize(prunedList);
14    // Send to the clients that need with a latency requirement of 100
      ms.
15    send_to_partion_clients(serializedPruneL, pId);
16    // Send it up to the concat node, through the objects edge.
17    send_up(serializedPruneL, "objects")
18  }
19 };
```

Figure A.3: Connected cars—Prune application component.

```
class DbscanNode : ApplicationComponent
{
  // Auxiliary functions
  set<string> getCurrentCars(partitionId pId);
  array<ObservationsWithId> getLastObservations(partitionId pId);
  // Override default handlers
  void on_send_up(msg m, partionId pId) override;
  void on_migration_end(partitionId pId) override;
};
```

Figure A.4: Connected cars—Dbscan application component interface.

```
set<string> DbscanNode::getCurrentCars(partitionId pId)
{
  // current time
  currentTime = getCurrentTime();
  // create time range
  timeRage = TimeRange(currentTime - maxWaitTime, currentTime);
  // Query the spatio-temporal object store
  allCarsInLastWindow = get("cars", pId, timeRange);
  // Return a set of the cars that the component should wait for
  return toSet(allCarsInLastWindow);
}

array<ObservationsWithId> getLastObservations(partitionId pId)
{
  // current time
  currentTime = getCurrentTime();
  // create time range
  timeRage = TimeRange(currentTime - maxWaitTime, currentTime);
  // Query the spatio-temporal object store
  observationsSinceLastTime = get("observations", pId, timeRange);
  // Return a set of the cars that the component should wait for
  return toArray(observationsSinceLastTime);
}
```

Figure A.5: Connected cars—Dbscan application component: auxiliary functions.

```
// Handler for receiving messages from a car
void DbscanNode::on_send_up(msg m, partionId pId)
{
  // Deserialize message
  input = m.deserialize();
  // Save that this car with the time when the data was generated
  put_object(input.carId, "cars", pId, input.messageTime);
  put_object(input.observationsWithId, "observations", pId, input.
    messageTime);
  // Get Current cars
  curentCars = getCurrentCars(pId);
  // Get latest observations in time window
  observations = getLastObservations(pId);
  // Get set of cars in those observatiosn
  carsInObservations = getCarsInObservations(observations);
  // Check if all cars that we are waiting are in the observations
  if(carsInObservations.contains(currentCars))
  {
    // All cars are present, then we can apply dbscan
    objectsList = dbscan(observations);
    // Serialize the object
    serializedObj = serialize(objectsList);
    // Send to the prune node
    send_up(serializedObj, "objects")
  }
  else
  {
    // Wait for the missing cars to come
    // More realistic implementations would also have a timeout
    // To apply the algorithm even if not all cars are presented.
  }
}
```

Figure A.6: Connected cars—Dbscan application component: “on send up”.

```
void DbscanNode::on_migration_end(partitionId pId)
{
  // Forcefully fetch the last window of cars
  currentCars = getCurrentCars(pId);
  // Forcefully fetch the last window of observations
  observations = getLastObservations(pId);
}
```

Figure A.7: Connected cars—Dbscan application component: “on migration end”.

```
string getSubRegionId(GPS client)
{
  // A simplified example of how would the data be processed
  if(centerOfCity.near(client, 10))
  {
    // If the client is within 10 km of the center of the city return 0
    return "0";
  }
  // If it is further away than 10 km return 1
  return "1";
}

string get_spatial_affinity_identifier(GPS client, string regionType)
{
  // Depending on the region type use a different partition id matching
     function
  if(regionType == "subregion")
  {
    identifier = getSubRegionId(client);
  }
  else if(regionType == "region")
  {
    identifier = getRegionId(client);
  }
  return identifier;
}
```

Figure A.8: Connected cars—Spatial affinity function.

REFERENCES

[1] GDPR.eu, *What is gdpr, the eu's new data protection law?* https://gdpr.eu/what-is-gdpr/, 2022.

[2] *Azure geographies*, https://azure.microsoft.com/en-us/global-infrastructure/geographies/, 2021.

[3] Microsoft, *Azure: Public Multi-Access Edge Compute*, https://azure.microsoft.com/en-us/solutions/public-multi-access-edge-compute-mec/#overview, 2022.

[4] J. Le Feuvre, J.-M. Thiesse, M. Parmentier, M. Raulet, and C. Daguet, "Ultra high definition hevc dash data set," in *Proceedings of the 5th ACM Multimedia Systems Conference*, ser. MMSys '14, Singapore, Singapore: Association for Computing Machinery, 2014, pp. 7–12.

[5] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, London, United Kingdom: Association for Computing Machinery, 2015, pp. 183–197.

[6] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, *et al.*, "Hydra: A federated resource manager for data-center scale analytics," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 177–192.

[7] *Azure Edge Zone preview*, https://docs.microsoft.com/en-us/azure/networking/edge-zones-overview, 2021.

[8] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu, "From cloud to edge: A first look at public edge platforms," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21, Virtual Event: Association for Computing Machinery, 2021, pp. 37–53.

[9] ——, "From cloud to edge: A first look at public edge platforms," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21, Virtual Event: Association for Computing Machinery, 2021, pp. 37–53.

[10] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 167–167.

[11] *KubeEdge: A kubernetes native edge computing framework*, https://kubeedge.io/en/, 2022.

[12] *Openwhisk: Open source serverless cloud platform*, https://openwhisk.apache.org/, 2022.

[13] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[14] *AWS: Regions and availability zones*, https://aws.amazon.com/about-aws/global-infrastructure/regions_az/, 2021.

[15] *Microsoft partners with the industry to unlock new 5g scenarios with Azure Edge Zones*, https://azure.microsoft.com/en-us/blog/microsoft-partners-with-the-industry-to-unlock-new-5g-scenarios-with-azure-edge-zones/, 2021.

[16] *Microsoft Azure: Ultra-low-latency edge computing*, https://azure.microsoft.com/en-us/solutions/low-latency-edge-computing/, 2021.

[17] *VaporIO: a nationwide platform for edge*, https://www.vapor.io, 2020.

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[19] *Apache Storm*, https://storm.apache.org/, 2021.

[20] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, Mar. 2004.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007, pp. 59–72.

[22] E. Saurez, H. Gupta, A. Daglis, and U. Ramachandran, "OneEdge: An efficient control plane for geo-distributed infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21, Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 182–196.

[23] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applica-

tions in the fog," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 258–269.

[24] M. Satyanarayanan, "Accessing information on demand at any location. mobile information access," *IEEE personal Communications*, vol. 3, no. 1, pp. 26–33, 1996.

[25] M. Weiser, "Some computer science issues in ubiquitous computing," *Commun. ACM*, vol. 36, no. 7, pp. 75–84, Jul. 1993.

[26] Wikipedia, *Pokemon go*, https://en.wikipedia.org/wiki/Pok%C3%A9mon_Go, 2020.

[27] S. Hayat, R. Jung, H. Hellwagner, C. Bettstetter, D. Emini, and D. Schnieders, "Edge computing in 5g for drone navigation: What to offload?" *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 2571–2578, 2021.

[28] A. Zihao Zhu, N. Atanasov, and K. Daniilidis, "Event-based visual inertial odometry," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Honolulu, HI: IEEE, 2017, pp. 5391–5399.

[29] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, 154–es, 1996.

[30] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, "Orleans: Distributed virtual actors for programmability and scalability," *MSR-TR-2014–41*, 2014.

[31] 3GPP, "Study on traffic characteristics and performance requirements for AI/ML model transfer in 5GS," 3rd Generation Partnership Project (3GPP), Technical Report 22.874, Dec. 2021, Version 18.2.0.

[32] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *kdd*, vol. 96, 1996, pp. 226–231.

[33] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 69–84.

[34] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21, Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 138–152.

[35] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics

platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[36] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.

[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[38] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 351–364.

[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[40] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 285–300.

[41] Z. Wang, H. Li, Z. Li, X. Sun, J. Rao, H. Che, and H. Jiang, "Pigeon: An effective distributed, hierarchical datacenter job scheduler," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 246–258.

[42] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, 2011, pp. 22–22.

[43] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 485–497.

[44] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 499–510.

[45] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*, 2015, pp. 97–110.

[46] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 373–377.

[47] *Google or-tools*, https://developers.google.com/optimization, 2022.

[48] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "TetriSched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[49] Cloud Native Computing Foundation, *Kubernetes*, https://github.com/kubernetes/kubernetes, 2022.

[50] *Kubernetes: Production-grade container orchestration*, https://kubernetes.io/, 2022.

[51] *Kubernetes Documentation: Kubernetes Components*, https://kubernetes.io/docs/concepts/overview/components/, 2022.

[52] *Kubernetes pods*, https://kubernetes.io/docs/concepts/workloads/pods/, 2022.

[53] *Kubernetes objects*, https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, 2022.

[54] *Etcd watch*, https://etcd.io/docs/v3.4/dev-guide/interacting_v3/#watch-key-changes, 2022.

[55] *Kubernetes API*, https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/, 2022.

[56] *Kubernetes scheduler: Eviction*, https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, 2022.

[57] *Kubernetes scheduler framework*, https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/, 2022.

[58] *Kubernetes: Resource management for pods and containers*, https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, 2022.

[59] *Kubernetes API reference docs: Affinity and anti-affinity*, https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/#affinity-v1-core, 2022.

[60] *Mongodb: Geospatial queries*, https://docs.mongodb.com/manual/geospatial-queries/, 2022.

[61] L. Manual, *Tc - traffic control*, https://linux.die.net/man/8/tc, 2020.

[62] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA: USENIX, 2019, p. 21.

[63] Kubernetes Multicluster SIG, *KubeFed*, https://github.com/kubernetes-sigs/kubefed, 2022.

[64] *Kubefed: Concepts*, https://github.com/kubernetes-sigs/kubefed/blob/master/docs/concepts.md, 2022.

[65] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 14–26, Jul. 2009.

[66] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[67] *Protocol Buffers (protobuf)*, https://developers.google.com/protocol-buffers, 2022.

[68] Docker, *Docker engine overview*, https://docs.docker.com/engine/, May 2020.

[69] *Rocksdb*, http://rocksdb.org/, Accessed: 2016-01-12.

[70] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz, "Sumo - simulation of urban mobility: An overview," in *in SIMUL 2011, The Third International Conference on Advances in System Simulation*, 2011, pp. 63–68.

[71] C. P. Wright and E. Zadok, "Unionfs: Bringing File Systems Together," *Linux Journal*, vol. 2004, no. 128, pp. 24–29, Dec. 2004.

[72] M. Haklay and P. Weber, "OpenStreetMap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, pp. 12–18, Oct. 2008.

[73] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[74] W. Milliken, T. Mendez, and D. C. Partridge, *Host Anycasting Service*, RFC 1546, Nov. 1993.

[75] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.

[76] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "The case for rack-out: Scalable data serving using rack-scale systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, ACM, 2016, pp. 182–195.

[77] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "Videoedge: Processing camera streams using hierarchical clusters," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2018, pp. 115–131.

[78] M. Hoffmann, A. Lattuada, J. Liagouris, V. Kalavri, D. Dimitrova, S. Wicki, Z. Chothia, and T. Roscoe, "SnailTrail: Generalizing critical paths for online analysis of distributed dataflows," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 95–110.

[79] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight recoverable virtual memory," *ACM Trans. Comput. Syst.*, vol. 12, no. 1, pp. 33–57, Feb. 1994.

[80] R. Haskin, Y. Malachi, and G. Chan, "Recovery management in quicksilver," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 82–108, 1988.

[81] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[82] CoreOS, *Improving kubernetes scheduler performance*, https://web.archive.org/web/20201108105259/https://coreos.com/blog/improving-kubernetes-scheduler-performance.html, 2020.

[83] Docker, *Runtime options with memory, cpus, and gpus*, https://docs.docker.com/config/containers/resource_constraints/, 2020.

[84] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, *Crawdad data set epfl/mobility (v. 2009-02-24)*, 2009.

[85] City-Data, *FCC Registered Cell Phone and Antenna Towers in San Francisco, California*, https://www.city-data.com/towers/cell-San-Francisco-California.html, 2021.

[86] O. Robotics, *Robot Operating System(ROS)*, https://www.ros.org/about-ros/, 2021.

[87] R. Jung, G. Rischner, E. Allak, A. Hardt-Stremayr, and S. Weiss, *Aau synthetic ros dataset for vio*, version V1, University of Klagenfurt, Zenodo, May 2020.

[88] Z. Zhang, S. Wang, Y. Hong, L. Zhou, and Q. Hao, "Distributed dynamic map fusion via federated learning for intelligent networked vehicles," in *2021 IEEE In-*

ternational Conference on Robotics and Automation (ICRA), Xi'an, China: IEEE, 2021, p. 12.

[89] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, Moutain View, CA: Journal of Machine Learning Research, 2017, pp. 1–16.

[90] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 751–766.

[91] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[92] *Apache Beam*, https://beam.apache.org/, 2022.

[93] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for Mobile/Cloud applications," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 97–112.

[94] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[95] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08, Vancouver, Canada: ACM, 2008, pp. 1123–1134.

[96] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, Budapest, Hungary: Association for Computing Machinery, 2018, pp. 236–252.

[97] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 696–707, 2018.

[98] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, “Spanedge: Towards unifying stream processing over central and near-the-edge data centers,” in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, 2016, pp. 168–178.

[99] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr., T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens, “Sla guarantees for cloud services,” *Future Generation Computer Systems*, vol. 54, pp. 233–246, 2016.

[100] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta, “Henge: Intent-driven multi-tenant stream processing,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’18, Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 249–262.

[101] *StarlingX*, https://www.starlingx.io/, 2022.

[102] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, “Fogernetes: Deployment and management of fog computing applications,” in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–7.

[103] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, “Mck8s: An orchestration platform for geo-distributed multi-cluster environments,” in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021, pp. 1–10.

[104] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, “Geo-distributed efficient deployment of containers with kubernetes,” *Computer Communications*, vol. 159, pp. 161–174, 2020.

[105] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, “Decentralized self-adaptation for elastic data stream processing,” *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018.

[106] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair Scheduling for Distributed Computing Clusters,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP ’09, Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 261–276.

[107] W. Dai, L. Qiu, A. Wu, and M. Qiu, “Cloud infrastructure resource allocation for big data applications,” *IEEE Transactions on Big Data*, vol. 4, no. 3, pp. 313–324, 2018.

[108] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types,” in *8th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 11), 2011.

[109] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 455–466.

[110] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 323–336.

[111] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: Wan-aware optimization for analytics queries," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 435–450.

[112] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," in *NSDI*, 2010.

[113] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17, San Jose, California: Association for Computing Machinery, 2017.

[114] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 26–35, 2017.

[115] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020.

[116] A. Yousefpour, A. Patil, G. Ishigaki, I. Kim, X. Wang, H. C. Cankaya, Q. Zhang, W. Xie, and J. P. Jue, "Fogplan: A lightweight qos-aware dynamic fog service provisioning framework," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 5080–5096, 2019.

[117] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '16, Irvine, California: Association for Computing Machinery, 2016, pp. 69–80.

[118] C. Zhu, J. Tao, G. Pastor, Y. Xiao, Y. Ji, Q. Zhou, Y. Li, and A. Ylä-Jääski, "Folo: Latency and quality optimized task allocation in vehicular fog computing," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4150–4161, 2019.

[119] L. Lin, X. Liao, H. Jin, and P. Li, "Computation offloading toward edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1584–1607, 2019.

[120] G. Janßen, I. Verbitskiy, T. Renner, and L. Thamsen, "Scheduling stream processing tasks on geo-distributed heterogeneous resources," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 5159–5164.

[121] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-aware component composition for distributed stream processing systems," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, ser. Middleware '06, Melbourne, Australia: Springer-Verlag, 2006, pp. 322–341.

[122] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018.

[123] C. Cicconetti, M. Conti, and A. Passarella, "Low-latency distributed computation offloading for pervasive environments," in *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2019, pp. 1–10.

[124] ——, "A decentralized framework for serverless edge computing in the internet of things," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 2166–2180, 2021.

[125] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021.

[126] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[127] Prometheus, *The prometheus monitoring system and time series database*, https://github.com/prometheus/prometheus, 2020.

[128] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: Design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[129] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, and L. Foschini, "Dargos: A highly adaptable and scalable monitoring architecture for multi-tenant clouds," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2041–2056, 2013, Including Special sections: Advanced Cloud Monitoring Systems & The fourth IEEE International Conference on e-Science 2011 — e-Science Applications and Tools & Cluster, Grid, and Cloud Computing.

[130] A. AWS, *Amazon cloudwatch: Observability of your aws resources and applications on aws and on-premises*, https://aws.amazon.com/cloudwatch/, 2022.

[131] M. Andreolini, M. Colajanni, M. Pietri, and S. Tosi, "Adaptive, scalable and reliable monitoring of big data on clouds," *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 67–79, 2015, Special Issue on Scalable Systems for Big Data Management and Analytics.

[132] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Jcatascopia: Monitoring elastically adaptive applications in the cloud," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 226–235.

[133] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *[1988] Proceedings. The 8th International Conference on Distributed*, 1988, pp. 366–373.

[134] Q. Guo, Y. Li, T. Liu, K. Wang, G. Chen, X. Bao, and W. Tang, "Correlation-based performance analysis for full-system mapreduce optimization," in *2013 IEEE International Conference on Big Data*, 2013, pp. 753–761.

[135] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, *et al.*, "The design of the borealis stream processing engine.," in *CIDR*, vol. 5, 2005, pp. 277–289.

[136] A. Brogi, S. Forti, and M. Gaglianese, "Measuring the fog, gently," in *International Conference on Service-Oriented Computing*, Springer, 2019, pp. 523–538.

[137] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Admin: Adaptive monitoring dissemination for the internet of things," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[138] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Run-time adaptation of data stream processing systems: The state of the art," *ACM Comput. Surv.*, Jan. 2022.

[139] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[140] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 399–410.

[141] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proceedings of the Sixth*

ACM Symposium on Cloud Computing, ser. SoCC '15, Kohala Coast, Hawaii: Association for Computing Machinery, 2015, pp. 276–287.

[142] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*, Mumbai, India: Association for Computing Machinery, 2014, pp. 13–22.

[143] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "Tcep: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms," in *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '18, Hamilton, New Zealand: Association for Computing Machinery, 2018, pp. 136–147.

[144] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: Autoscaling for real-time stream analytics," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3338–3352, 2017.

[145] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems and Software*, vol. 127, pp. 205–216, 2017.

[146] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leung, "Dynamic service migration and workload scheduling in edge-clouds," *Performance Evaluation*, vol. 91, pp. 205–228, 2015, Special Issue: Performance 2015.

[147] S. Wang, R. Urgaonkar, K. Chan, T. He, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," in *Communications (ICC), 2015 IEEE International Conference on*, IEEE, 2015, pp. 5504–5510.

[148] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran, "Mcep: A mobility-aware complex event processing system," *ACM Trans. Internet Technol.*, vol. 14, no. 1, Aug. 2014.

[149] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 783–798.

[150] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading for pervasive computing," *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 66–73, 2004.

[151] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, “You can teach elephants to dance: Agile vm handoff for edge computing,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC ’17, San Jose, California: Association for Computing Machinery, 2017.

[152] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino, “Companion fog computing: Supporting things mobility through container migration at the edge,” in *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2018, pp. 97–105.

[153] P. S. Junior, D. Miorandi, and G. Pierre, “Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes,” in *ICFEC 2022-6th IEEE International Conference on Fog and Edge Computing*, 2022.

[154] L. Ma, S. Yi, and Q. Li, “Efficient service handoff across edge servers via docker container migration,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC ’17, San Jose, California: Association for Computing Machinery, 2017.

[155] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, “Migcep: Operator migration for mobility driven distributed complex event processing,” in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS ’13, Arlington, Texas, USA: ACM, 2013, pp. 183–194.

[156] M. Hoffmann, A. Lattuada, and F. McSherry, “Megaphone: Latency-Conscious State Migration for Distributed Streaming Dataflows,” *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 1002–1015, May 2019.

[157] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating scale out and fault tolerance in stream processing using operator state management,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’13, New York, New York, USA: Association for Computing Machinery, 2013, pp. 725–736.

[158] W. J. Dally, “The future of computing: Domain-specific accelerators,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’52)*, 2019.

[159] N. Arora, T. Starner, and G. D. Abowd, “Saturn: An introduction to the internet of materials,” *Commun. ACM*, vol. 63, no. 12, pp. 92–99, Nov. 2020.

[160] Hadoop, *Hadoop: Rack awareness*, https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/RackAwareness.html, 2022.

[161] E. Saurez, B. Balasubramanian, R. Schlichting, S. P. Narayanan, B. Tschaen, Z. Huang, and U. Ramachandran, "A Drop-in Middleware for Serializable DB Clustering across Geo-distributed Sites," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3340–3353, 2020.

[162] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[163] D. Kang, P. Bailis, and M. Zaharia, "Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 533–546, 2019.