DESIGNING SCALABLE MECHANISMS FOR GEO-DISTRIBUTED PLATFORM SERVICES IN THE PRESENCE OF CLIENT MOBILITY

A Dissertation
Presented to
The Academic Faculty

By

Harshit Gupta

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

December 2022

DESIGNING SCALABLE MECHANISMS FOR GEO-DISTRIBUTED PLATFORM SERVICES IN THE PRESENCE OF CLIENT MOBILITY

Thesis committee:

Professor Umakishore Ramachandran, Advisor
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

Professor Ahmed Saeed
School of Computer Science
Georgia Institute of Technology

Dr. Abhigyan Sharma
Meta

Date approved: August 26th, 2022

To my mother, my late father, and my sister, for all their love, support and encouragement.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Umakishore Ramachandran, for his continuous support and motivation during my path as a PhD student. I routinely claim to be one of the luckiest PhD students ever to have an advisor like Prof. Kishore. He has helped me constantly both by steering my research efforts in the right direction and by helping me maneuver out of technical roadblocks. More than that, his kind and understanding attitude toward his students has made my PhD journey much more enjoyable than I would have imagined. I will forever be grateful and indebted to him for the support he has provided to me.

I would like to thank the members of my dissertation committee for taking their valuable time to be on my committee and providing feedback that has significantly improved my dissertation. I would specifically like to thank Professor Ada Gavrilovska for mentoring me in my earlier years as a PhD, when we worked on a project together. She has always encouraged me to aim higher in my academic journey. Next, I would like to thank Prof. Alexandros Daglis, for his support and contribution toward the OneEdge project. His inputs helped us significantly improve the paper and get it accepted for publication. I would like to thank Dr. Abhigyan Sharma for his continued encouragement and help that he has provided ever since I have known him, especially his guidance for the job search. Finally, I would like to thank Prof. Ahmed Saeed for helping me improve my dissertation during my proposal.

As with any journey, fellow travelers are crucial to make it a good experience. I express a special thanks to my colleague Enrique Saurez, with whom I have been working from day one of my PhD to almost the very end. Not only have we collaborated extensively for research, we have also shared a lot of good times outside school. He has also selflessly helped me out on a number of occasions and made life a lot easier for me. In the same vein, I would like to thank my colleagues Adam Hall and Zhuangdi Xu, for not only collaborating me and helping me with my dissertation, but also being there as close friends for almost the entirety of my stay at Georgia Tech. I am grateful for all my wonderful colleagues at the Embedded Pervasive Laboratory (EPL) – Tyler Landle, Anirudh Sarma, Manasvini Sethuraman, Alan Nussbaum, Jinsun Yoo, Difei Cao, Ashish Bijlani and Wonhee Cho. I am also grateful to my collaborators from the University of Stuttgart – Prof. Kurt Rothermel, Ruben Mayer, Henriette Roger, and Sukanya Bhowmik, for the interesting work we did with each other. I am also indebted to my external collaborator from AT&T Alex Zelezniak for lending his immense expertise to my projects.

On a personal note, I am thankful beyond words for my family – my mother and sister and my late father – for their never ending love and encouragement on every step of the way. I am indebted to my very dear friend Nivya Sharma, who has been a part of my life for the past year and a half and helped me maneuver through the most difficult and hectic part of my PhD. Last but not least, I would like to thank my friends Ipsita and Shubhayan who have helped me throughout my stay at Georgia Tech with anything I needed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

AoI	Area of Interest
CDF	Cumulative Distribution Function
HUD	Heads-up Display
LiDAR	Light Detection and Ranging
PTZ	Pan-Tilt-Zoom
RNIS	Radio Network Information Service
RTT	Round Trip Time
UAV	Unmanned Aerial Vehicle
VM	Virtual Machine
WAN	wide-area network

SUMMARY

Situation-awareness applications require low-latency response and high network bandwidth, hence benefiting from geo-distributed Edge infrastructures. The developers of these applications typically rely on several platform services, such as Kubernetes, Apache Cassandra and Pulsar, for managing their compute and data components across the geo-distributed Edge infrastructure. Situation-awareness applications impose peculiar requirements on the compute and data placement policies of the platform services. Firstly, the processing logic of these applications is closely tied to the physical environment that it is interacting with. Hence, the access pattern to compute and data exhibits strong spatial affinity. Secondly, the network topology of Edge infrastructure is heterogeneous, wherein communication latency forms a significant portion of the end-to-end compute and data access latency. Therefore, the placement of compute and data components has to be cognizant of the spatial affinity and latency requirements of the applications. However, clients of situation-awareness applications, such as vehicles and drones, are typically mobile – making the compute and data access pattern dynamic and complicating the management of data and compute components. Constant changes in the network connectivity and spatial locality of clients due to client mobility results in making the current placement of compute and data components unsuitable for meeting the latency and spatial affinity requirements of the application. Constant client mobility necessitates that client location and latency offered by the platform services be continuously monitored to detect when application requirements are violated and to adapt the compute and data placement. The control and monitoring modules of off-the-shelf platform services do not have the necessary primitives to incorporate spatial affinity and network topology awareness into their compute and data placement policies. The spatial location of clients is not considered as an input for decision-making in their control modules. Furthermore, they do not perform fine-grained end-to-end monitoring of observed latency to detect and adapt to performance degradations due to client mobility.

This dissertation presents three mechanisms that inform the compute and data placement policies of platform services, so that application requirements can be met.

- M1: Dynamic Spatial Context Management for system entities – clients and data and compute components – to ensure spatial affinity requirements are satisfied.
- M2: Network Proximity Estimation to provide topology-awareness to the data and compute placement policies of platform services.
- M3: End-to-End Latency Monitoring to enable collection, aggregation and analysis of per-application metrics in a geo-distributed manner to provide end-to-end insights into application performance.

The thesis of our work is that the aforementioned mechanisms are fundamental building blocks for the compute and data management policies of platform services, and that by incorporating them, platform services can meet application requirements at the Edge. Furthermore, the proposed mechanisms can be implemented in a way that offers high scalability to handle high levels of client activity. We demonstrate by construction the efficacy and scalability of the proposed mechanisms for building dynamic compute and data orchestration policies by incorporating them in the control and monitoring modules of three different platform services. Specifically, we incorporate these mechanisms into a topic-based publish-subscribe system (ePulsar), an application orchestration platform (OneEdge), and a key-value store (FogStore). We conduct extensive performance evaluation of these enhanced platform services to showcase how the new mechanisms aid in dynamically adapting the compute/data orchestration decisions to satisfy performance requirements of applications.

CHAPTER 1
INTRODUCTION

Situation-awareness applications sense the physical environment, extract actionable information from it, and perform actions based on the extracted information. They perform critical tasks such as navigation control for Unmanned Aerial Vehicles (UAVs), autonomous vehicle control, and Pan-Tilt-Zoom (PTZ) tuning for connected cameras, where response time of the application as perceived by the end-client (UAV or connected camera) should be low for ensuring correct functionality. Besides, sensors such as cameras and Light Detection and Rangings (LiDARs) generate streams with a high data rate, due to which these applications require high network bandwidth. Hence, instances of such applications need to be deployed in close network proximity from end-clients instead of Cloud datacenters to ensure that network traversal through the wide-area network (WAN) does not adversely impact response time or impose limits on available bandwidth. Edge computing has gained prominence as a computing paradigm that utilizes computational and storage resources at the edge of the network, thereby allowing application instances to be deployed across a continuum of resources ranging from access networks to datacenters [1]. Utilizing Edge infrastructure for hosting situation-awareness applications would allow them to achieve predictable and low response times.

Managing situation-awareness applications requires deployment and scaling of application instances based on client demand, necessitating the use of an application orchestrator such as Kubernetes. These applications possess a number of communicating entities which exchange information that is integral to their functionality, naturally lending itself to using a publish-subscribe system, such as Apache Pulsar, to enable efficient communication. These applications also need to store state that is used to guide their future actions. They need access to a database, such as Apache Cassandra, to store and query application state. Access to platform services is in the critical path of the application logic of our target applications. Hence, platform service instances need to be deployed on Edge resources to avoid high communication overhead when accessing them. Situation-awareness applications require that access to platform services does not introduce significant overhead such that the application's response time is affected. Furthermore, these applications have a strong dependence on spatial location for mapping end-devices to application instances and defining communication and data access patterns.

1.1 Problem Statement

Although the data-plane of contemporary platform services offer intuitive semantics and high performance, their control policies for managing compute and data components are designed and optimized for operation in datacenters. The latency and spatial affinity requirements of situation-awareness applications make these policies unsuitable for the Edge infrastructure. Firstly, the network topology of Edge infrastructure is heterogeneous, with high variability in network latency between Edge sites unlike datacenters, where nodes are connected together by a fast and low-latency interconnect. Hence, network latency between clients and platform service nodes, and between platform service nodes is assumed to be negligible in cloud-based platform service deployments. Using such network-latency-agnostic control policies in platform services deployed at the Edge would result in high overheads due to network latency in the critical path of applications. Secondly, contemporary platform services do not consider client location as a first-class citizen for making control decisions. Therefore, it becomes the application developer's burden to ensure that client-to-application mapping and communication between system components are done in a location-aware manner. Therefore, platform services should perform data and compute placement by taking into account the latency and spatial affinity requirements of the applications, so that application requirements can be met and developer burden can be minimized.

However, clients of situation-awareness applications, such as autonomous vehicles and drones, are inherently mobile, which further complicates the decision-making of compute and data components. Client mobility creates significant dynamism in the input workload. The network connectivity of clients changes as a result of mobility, which frequently results in the application instance currently serving a given client unsuitable for meeting its low latency requirement. Similarly, a change in client location results in the client interacting with a different set of clients or access data corresponding to the new spatial locality it is in. Therefore the compute and data placement decisions need to be dynamically made and updated commensurate with client mobility. Furthermore, client mobility also results in the occurrence of skews in workload distribution which could create performance hotspots in specific platform service nodes resulting in higher latency overhead. To account for these sources of dynamism, the platform services need to monitor client location and all latency overheads and make reconfiguration decisions in the case of violation of spatial affinity or response time requirements. The dynamic management of data and compute components should be scalable to support the large number of clients and the high frequency of such adaptations due to mobility.

1.2 Thesis Statement

In order to solve the challenges faced when designing control policies for Edge-based platform services, this dissertation proposes three mechanisms that provide relevant information to the control and monitoring components of a platform service, such that it can take actions and continuously satisfy client's performance requirements.

- M1: Dynamic Spatial Context Management for system entities – clients and data and compute components – to ensure spatial affinity requirements are satisfied. Our proposed mechanism uses the spatial context of clients, data and compute components as a new form of input to manage access to application instances and data items.
- M2: Network Proximity Estimation to provide topology-awareness to the data and compute placement policies of platform services.
- M3: End-to-End Latency Monitoring to enable collection, aggregation and analysis of per-application metrics in a geo-distributed manner to provide end-to-end insights into application performance. End-to-end analysis of application latency enables platform services to detect and ascertain the cause of a violation of performance requirements.

Thesis Statement: The proposed mechanisms working in tandem are the fundamental building blocks for the control and monitoring components of platform services, catering to the high efficiency, scalability, and resource frugality needs of these services on Edge infrastructures enabling the realization of geo-distributed and mobile situation-awareness applications.

1.3 Contributions

This dissertation presents three novel mechanisms that aid the decision-making of compute and data components on geo-distributed Edge infrastructure[1]. To this end, we make the following contributions.

[1]The mechanisms M1 and M3 were co-invented by myself and my colleague Enrique Saurez [2] from the School of Computer Science at Georgia Institute of Technology. However, the formalization and design-space exploration of M1 and M3 are solely the contribution of my research.

- We formalize the proposed mechanisms by defining the interface exposed to the platform services. For each proposed mechanism, we highlight its necessity and demonstrate the utility of the interface in the context of an example control policy of a platform service.

- We perform a design-space exploration of the proposed mechanisms with respect to their application in a geo-distributed platform service. The candidate design choices are evaluated in terms of their efficiency, scalability and resource frugality. For each mechanism, we pick the design choice that outperforms the others in meeting the above objectives.

The applicability of the proposed mechanisms is then demonstrated by using them to build control policies for three edge-centric platform services and evaluating the observed performance of typical situation awareness applications. These platform services are described as follows.

- **OneEdge**, an application orchestration platform that performs placement of application components using the Network Proximity Estimation mechanism to meet the application's response time requirements. Clients are mapped to application instances based on their spatial affinity requirements using the Dynamic Spatial Context Management mechanism. **OneEdge** leverages the End-to-End Latency Monitoring mechanism for continuous monitoring of observed response times with custom policies detects violations of application requirements and triggers migration of client to a different application instance. [2]

- **ePulsar** , a topic-based publish-subscribe system that performs topic (data) placement among brokers on Edge sites using the Network Proximity Estimation mechanism to satisfy end-to-end message delivery latency requirements. **ePulsar** uses the End-to-End Latency Monitoring mechanism to record the end-to-end message delivery latency of each topic. When the latency threshold for a topic is violated, a migration of that topic to a different broker is triggered.

- **FogStore**, a key-value store, that meets a developer-specified tradeoff between latency, consistency and fault tolerance. Developers specify the spatio-temporal context of data items, and **FogStore** uses the Dynamic Spatial Context Management

[2]The implementation of **OneEdge**, along with the integration of mechanisms M1 and M3 was done jointly by myself and my colleague Enrique Saurez. The integration of mechanism M2 in OneEdge is solely my work, along with the experimental studies presented in this dissertation.

mechanism to determine the optimal data placement and consistency level for clients based on the data-item's context. **FogStore** is able to provide consistent access with low latency by exploiting the spatial-locality in data access patterns of applications.

1.4 Roadmap

The remainder of this document is structured as follows. Chapter 2 discusses the target application space, i.e., situation-awareness applications, including their general characteristics, specific examples and the requirements they pose on the platform services. The chapter also covers how these requirements can only be fulfilled by the introduction of new mechanisms into the control and monitoring modules of platform services. Next, Chapter 3 describes these mechanisms concretely, including the abstractions that they expose to control policies, and results from a set of experimental evaluations that quantify the possible improvement in control decisions if these mechanisms are used. Chapter 4 presents a design-space exploration of each of the mechanisms, wherein it quantitatively compares multiple designs for each mechanism in terms of efficiency and scalability. Chapters 5, 6 and 7 demonstrate the use of the proposed mechanisms in the control policies of **ePulsar**, **OneEdge** and **FogStore** respectively, as mentioned above. Chapter 8 presents the related work and their connection with this dissertation. Chapter 9 discusses the ideas and lessons learned by carrying out the research presented in this dissertation. Finally, Chapter 10 concludes the dissertation and presents directions for future research.

CHAPTER 2
SITUATION-AWARENESS APPLICATIONS

The proliferation of high-fidelity sensors such as cameras, LiDAR, etc., and the increasing access to sophisticated data analysis and machine learning models have enabled the emergence of novel situation-awareness applications. These applications interact with the physical environment by sensing and extracting relevant information and performing actions based on the extracted information. Examples of such applications include collaborative sensing for autonomous driving (that fuses the data perceived by sensors on multiple cars to improve driving) and collaborative PTZ tuning for a distributed camera network for tracking multiple targets simultaneously. These applications form the target use-cases for this dissertation's research. This chapter discusses the characteristics of these applications, the platform services that typical implementations of such applications would rely on, and the specific requirements that they impose on these platform services.

2.1 General Characteristics

Situation-awareness applications possess common characteristics which are highlighted below:

- **Sense-process-actuate control loop.** The applications of interest in this dissertation possess a sense-process-actuate control loop, wherein, applications sense the physical environment (e.g., through cameras or LiDAR sensors), extract information from the sensed data (e.g., the presence of a suspicious vehicle in a camera's view), and perform actions in response to events in the environment (e.g., moving the camera to better capture the target). This control loop functions at machine-perception speeds and does not involve human intervention in the critical path.
- **Interaction with physical world in proximity.** Situation-awareness applications function by sensing and acting upon the physical environment that is in close physical proximity to the end-clients, because the range of sensor and actuator devices is limited. Similarly, an instance of a situation-awareness application serving a particular end-client would only be interested in information about a subset of the environment in the vicinity of the client. Applications exhibit such spatial affinity to their physical

environment because they respond to changes in the environment and a particular application instance does not respond to changes happening very far away from the clients it is serving due to physical constraints. An example of an application with functionality tied to the physical environment is navigation for autonomous vehicles. The application uses camera or LiDAR sensor data to know about the immediate surroundings of the vehicle it is serving for collision avoidance and maneuvering, as well as data about traffic congestion in roads around the given vehicle to make routing decisions. This subset of the environment around the client forms its spatial context. Conversely, each data source (e.g., traffic congestion at a particular road segment) is also associated with a spatial context, which denotes the geographical area whose data would be of interest to a subset of the clients in that area.

- **Inter-client collaboration**. Multiple clients of the same application, such as autonomous cars, are expected to be operating in the same physical space. Clients that are in close proximity (and hence interacting with an overlapping geographical region) share spatial context and would benefit from sharing data among each other. Inter-client data sharing is also necessary for the correct functionality of the application, such as in the case of collaborative PTZ tuning of cameras. It is also useful for improving each client's perception of the environment by augmenting the sensing range of each client and alleviating occlusion, such as in the collaborative sensing for autonomous driving application. As described earlier, the set of clients that a given client interacts with depends on their spatial context, which is defined by the nature of the application, with some applications having a large and some having a small set of coordinating clients.

- **Client mobility.** Client devices in our target applications (such as drones and cars) are, by their very nature, mobile which leads to the environment that they are interacting with dynamic - thereby generating multiple types of dynamism in the system. Firstly, the workload characteristics vary over time, wherein the compute and network cost of processing a given client's sensor data changes temporally. An example of such a dynamism is when a vehicle running an obstacle detection application client moves from a rarely populated part of the city to a densely populated one, and the number of objects-of-interest in its sensor input increases substantially. An increase in the amount of useful data sensed by the vehicle increases the compute and network bandwidth requirements of processing and communicating data. Secondly, mobility changes the spatial context of the client, which then changes the set of clients it would coordinate with, as well as the set of data sources it would consume. For instance, in a collaborative perception application for driving assistance, the set of vehicles that a given vehicle collaborates with keeps changing continuously.

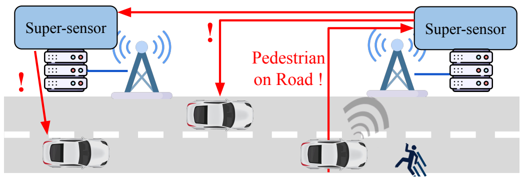
- **Temporal variation in workload.** Given that the target applications sense the physical environment, the workload served depends on the amount of activity (e.g., number of cars in view of camera) in the sensing range of the client. However, in a typical environment (e.g., an urban area) the amount of activity varies temporally and spatially. Hence, both static and mobile clients would sense different levels of activity over time, and generate variable levels of workload. This type of variation is distinct from the one discussed earlier (caused by client mobility), because this specific variation is present in the environment regardless of whether clients are static or mobile.

2.2 Exemplar Situation-Awareness Applications

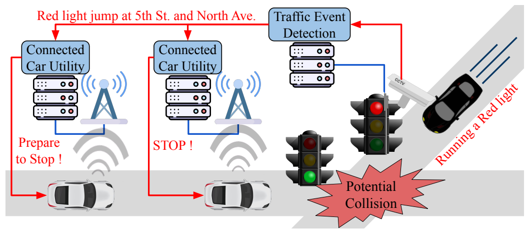
We now present concrete examples of situation-awareness applications that possess the characteristics discussed in Section 2.1. For each application, we first discuss its high-level functionality, and then present a candidate architecture along with the interactions of various components of the application.

2.2.1 Cooperative Sensing for Autonomous Driving

Autonomous driving vehicles are reliant on local on-board sensors such as LiDAR and stereo cameras for detecting objects on and around the street such as other vehicles or a pedestrian crossing the street. The control application of autonomous vehicles is responsible for taking an immediate action (such as braking or lane change) in response to unexpected and potentially dangerous events such as a jaywalking pedestrian or a vehicle jumping a red light. However, due to the inherent complexity of driving contexts in busy streets, it is possible that either the sensors are occluded by other objects in their field of view (Fig. 2.1a), or the sensing range of individual vehicles is not enough to capture the event (Fig. 2.1b) [3]. To better cope with the above scenarios, the fusion of sensed data from multiple nearby vehicles along with road-side infrastructure (such as CCTV cameras) can alleviate the issues of limited sensing range and occlusion. For instance, as shown in Fig. 2.1a and Fig. 2.1b, the sensor data from nearby vehicles and CCTV cameras are used to augment the local sensors on-board each vehicle. Receiving fused sensor data from mul-



(a) Detection of occluded pedestrians.



(b) Notification of a rogue vehicle jumping a red light.

Figure 2.1: Use-cases of cooperative sensing for Autonomous driving.

tiple vehicles enables each vehicle to gain a wider view of the current traffic situation and become aware of the traffic event sooner.

The fundamental components of the collaborative perception application and their functionality has been discussed and evaluated thoroughly by previous work[4, 3, 5]. We base the application design presented here on the F-Cooper [4] work done by Chen et al. Each vehicle is associated with a per-vehicle application component that processes raw sensor data perceived by local on-board sensors of that vehicle and extracts features about detected objects and their location with respect to the global coordinate space. The per-vehicle components corresponding to the set of physically close-by vehicles send streams of detected features to a fusion component. The fusion component performs feature matching among the objects reported by each per-vehicle component and de-duplicates multiple views of the same object. Through this de-duplication process, the fusion component merges the views of multiple close-by vehicles. Each vehicle is only interested in fused data corresponding to an area within roughly 600 meters from the vehicle's current location [6]. Thus a subset of the fused view is then sent back to each per-vehicle component, which relays it back to the vehicle – either to be displayed through a Heads-up Display (HUD) or for making navigation decision (in the case of autonomous vehicles).

The compute requirements of the fusion component is non-trivial [3], which necessitates multiple instances of the sensor fusion component serving vehicles distributed throughout a city. Boehme et al. [6] propose such a distributed architecture of the collaborative perception application, wherein the geographical space that the application serves is partitioned into several "regions" and all vehicles present in a specific region are served by the same fusion component. The division of a geographical space (e.g., a city) into regions is done so as to ensure uniform load distribution across the fusion components, however, that is out of the scope of this thesis.[1] In addition to the per-vehicle object features from

[1]Such a division can be done by taking into account historical levels of vehicular activity across space and

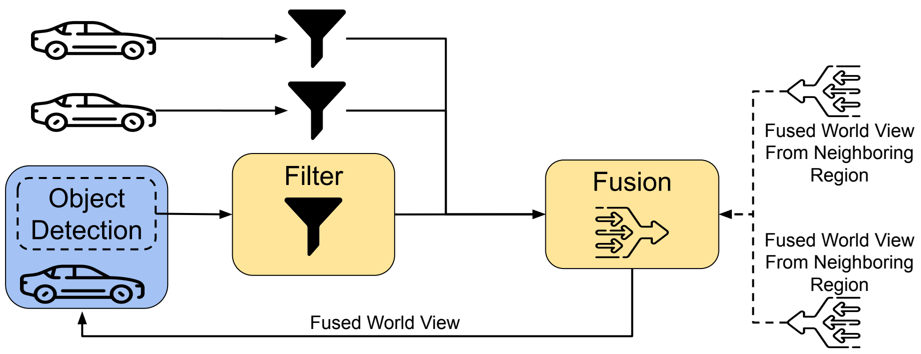

Filter
Object Detection
Fusion
Fused World View From Neighboring Region
Fused World View From Neighboring Region
Fused World View

Figure 2.2: An illustration of the components of the Collaborative Sensing for Autonomous Driving application.

the vehicles present in its region, each fusion component also receives fused data from geographically neighboring regions. The data from neighboring regions allows the fusion component to serve vehicles that are close to the periphery of its region, since the 600 meter *area-of-interest* of those vehicles might extend beyond the region's area.

The collaborative perception application continuously ingests data perceived about the physical environment in proximity to the vehicle client, processes it and returns the result back to the client at machine-perception speed. The response-time of this sense-process-actuate control loop needs to be very small (100ms [3]) so that the driver or autonomous driving agent can react to obstacles in real-time. The application involves collaboration between multiple clients, that are necessarily located in physical proximity to one another. The application also requires coordination between fusion components that serve neighboring regions because the area-of-interest of multiple clients that are close to each other overlap. Vehicles are inherently mobile, which creates dynamism in the compute and network requirements of the application, as well as the set of clients that are served by a given fusion component.

2.2.2 Collaborative Navigation Control in UAV Swarm

Applications such as large-scale road traffic monitoring or search-and-rescue are good candidates to apply UAVs given their flexible mobility and sensing capability. However the sensing range of cameras on an individual UAV makes it cumbersome to perform large-scale jobs such as traffic monitoring or search and rescue. Recently, the cost of UAVs has been declining and the communication infrastructure is becoming more omnipresent. Swarms of UAVs have been proposed to be used for road traffic monitoring [7], search and

ensuring that each region manager serves roughly uniform number of vehicles.

rescue [8] and surveillance [9]. The drones that are a part of the swarm need to navigate together in order to perform a task. For instance, in the case of road traffic monitoring, drones need to coordinate among themselves to cover the entire road segment monitored by them, while also avoiding collisions with obstacles such as poles or bridges, and other drones. In the following discussion, we will take up the use-case of road traffic monitoring to describe the application. However, the concepts generalize to other use-cases.

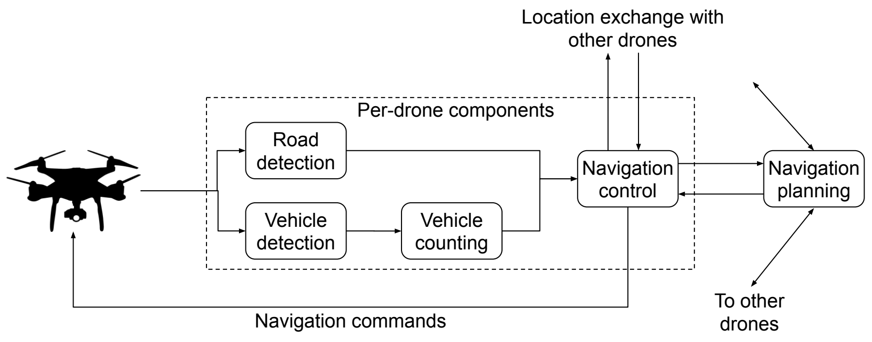


Figure 2.3: Schematic of the Collaborative UAV Navigation application.

Each drone is associated with a per-drone application component, which processes the sensor data from the drone's onboard sensors in real-time. The per-drone component extracts relevant information from the sensor stream, such as vehicle counts. The extracted information is communicated to the navigation planning component, which determines whether the drone should remain in the current region of the road segment, or start monitoring another region. Drones also communicate among themselves by sharing location updates and information about detected obstacles so that they can perform fine-grained navigation control.

2.2.3 Collaborative PTZ Tuning in a Distributed Camera Network

Smart cities are seeing CCTV cameras installed at a large number of locations to record and analyze unexpected events such as accidents and crimes. However, surveillance using CCTV cameras is largely done after an incident has occurred and the static deployment of cameras makes it difficult to fully capture the target objects. Online multi-camera object tracking has emerged as a recent development due to the proliferation of efficient machine learning models for computer vision [10]. Contemporary cameras are often equipped with Pan-Tilt-Zoom tuning capabilities which allow them to better track target objects. Furthermore, due to the high density of cameras, they can also work collaboratively in tracking multiple target objects [11].

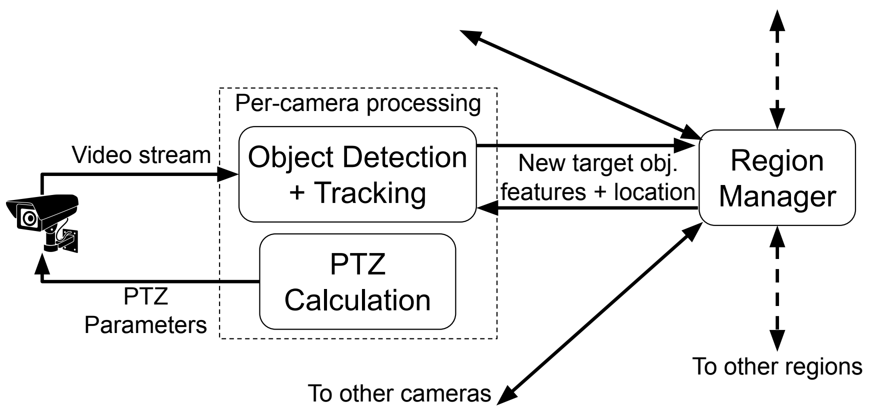


Figure 2.4: Schematic of Collaborative PTZ Tuning application for a Distributed Camera Network.

In order to achieve this goal, the cooperative vision application groups cameras into multiple regions and splits the application logic between a per-camera component and a per-region Region Manager (see Section 2.2.3). The per-camera processing component performs object detection and tracking and informs the region manager about the features of newly detected target objects and the current location of tracked target objects. It continuously tracks target objects by adjusting the PTZ parameters. The Region Manager consumes the current location of each target object and determines the assignment of cameras to target objects. In case a target object is about to leave the given region, the region manager informs the neighboring region(s) of the impending arrival of the target object.

The collaborative PTZ tuning application requires the processing of incoming video streams in real-time to determine PTZ tuning actions. It also experiences temporal fluctuations in workload depending on the density of vehicles in each camera's field of view, which varies significantly during the day. Furthermore, multiple region-level components coordinate among each other to perform vehicle tracking over large geographical areas.

2.3 Application Model

Having described the exemplar situation-awareness applications, we now abstract out the details of each application and try to model this class of applications. We characterize the applications in terms of their compute architecture, communication patterns, storage behavior and their functionality being tied to spatial context. We discuss each of these concepts in this section.

2.3.1 Computation Model

The compute model of our target applications comprises two main components:

- **Per-client component** for client-specific computation. This component is responsible for processing information generated by each client (e.g., an autonomous car or UAV) and providing input to higher layers of the application. The per-client component is specific to each application client and maintains the client's state.

- **Region-level component** for combining information extracted from multiple clients. Each region-level component is assigned a number of clients based on their geographical location. A given region-level component hosts coordination and collaboration tasks between all the clients mapped to it. In the event that coordination between two clients belonging to two different regions is required, their two corresponding region-level components communicate between each other and share information.

2.3.2 Communication Pattern

Communication among application entities in our target applications follow two primary patterns:

- **Communication between the per-client component and region-level component.** Clients share data with the corresponding region-level component for inter-client coordination and data-sharing within the region. Clients receive region-level information extracted from multiple clients. For example, in the collaborative PTZ tuning application, the per-camera component shares the current position of target objects currently assigned to it with the region manager, while receiving the locations of newly assigned target objects for it to track.

- **AoI based communication from a region-level component to per-client instances or other region-level components.** A particular data-item is expected to be received by an application component if the data-item represents an object or event that falls within the receiver's spatial context. This communication pattern can manifest itself in three ways: (1) region-level component to per-client component, as in the cooperative sensing application for autonomous driving, wherein a given vehicle receives the fused worldview not only from the region manager of the current region, but also from regions that overlap with the vehicle's AoI; (2) across clients which fall in each other's AoI, as in the collaborative collision avoidance module of the UAV swarm application, which requires that UAVs share their current location among each other so that they can be aware of other UAVs in their vicinity; (3) communication between region-level components for sharing information at the region level. For instance, in the collaborative PTZ tuning application, region managers share details about a target object that is about to leave one region toward the other.

2.3.3 Storage Requirements

Applications generate state about application execution and information sensed from the environment that need to be persisted to enable future queries. Examples of such state include the assignment of target objects to PTZ cameras, semi-permanent road closures or traffic incidents in the collaborative driving application. Situation-awareness applications typically execute range-queries on this state to select data-items that fall within a certain geographical area. Hence, these data-items are tagged with the geo-location of the entity they are describing. Data items in application's state are often read/updated by multiple entities. Applications expect a diverse set of consistency guarantees on data access based on their application logic and reliance on most recent version of data.

2.3.4 Spatial Affinity

Since the target applications interact with the physical environment, actions taken by the application logic are often dependent on events and information from the immediate physical proximity. Hence, both computation and communication patterns of the target applications exhibit spatial affinity. Spatial affinity is defined by the AoI of application clients and components, which represents the spatial area wherein other entities that the given entity directly interacts with are present. For instance, in the collaborative driving application, the AoI of a car contains all other objects in vicinity of the car whose position information is needed in real-time to avoid potential collisions. Spatial affinity plays an important role in all facets of the application.

- **Computation.** Clients in physical proximity to each other are likely within each other's AoI, and hence are grouped together and served by the same region-level application component.
- **Communication.** Communication patterns of the target applications are guided by

spatial affinity of clients. Each client communicates with other clients and region-managers that fall within or overlap with the given client's AoI.

- **Storage.** The state maintained and accessed by an application component pertains to objects and events in the subset of the physical environment that the application interacts with, and hence in physical proximity.

2.4 Functional Requirements

The target applications present a set of functional requirements on the underlying infrastructure.

- **Low latency requirement.** The sense-process-actuate control loop of situation-awareness applications needs to be processed with end-to-end latency under the application's predefined threshold. This requirement ensures that the clients are able to respond in real-time to changes in the physical environment.

- **Mobility-driven reconfiguration to maintain spatial affinity.** Clients are continuously mobile, and so the mapping of per-client to region-level components needs to be reconfigured based on the current location of clients, so that inter-client coordination and data-sharing is done only between clients that are in geographical proximity of each other.

2.5 Platform Services needed and Functional Requirements

The aforementioned target applications can be implemented using a combination of platform services. Platform services provide the necessary systems support with powerful semantics for the applications, so that the developers can focus on the core application logic.

2.5.1 Compute Orchestration

The target applications comprise of multiple distributed components, each with a specific functionality. These application components require appropriate resource allocation to cater to their specific computational requirements such that a low sense-process-actuate control-loop latency can be ensured. For the application's correctness, per-client components should be mapped to the right region-level component based on client's current location. This problem is further complicated by the mobility of clients. Client mobility necessitates the dynamic reassignment of per-client components to region-level components because the set of clients present in a given region keeps changing over time. Furthermore, the spatial distribution of clients varies with time, and therefore the workload at each application component. This necessitates dynamic updates to the resource allocation of application components to avoid under-utilization or over-utilization of resources.

Compute orchestration platform service handles all the above issues without requiring the developer's intervention. Given the latency and spatial affinity requirements of applications, the platform service will automatically deploy and reconfigure the resource allocation and connectivity of application components.

2.5.2 Publish-Subscribe Communication

The communication patterns in the target situation-awareness applications ranges from one-to-many (region-level component to per-client component) to many-to-one (per-client component to region-level component), to many-to-many patterns (among clients). Furthermore, given the continuous mobility of clients and the dependence of communication pattern on spatial affinity, the set of entities communicating with a given client changes over time. Implementing the communication subsystem for a given application would require maintaining a dynamically updated list of receivers for each data sender, ensuring reliable message delivery to all receivers, etc., which are challenging for application developers.

Publish-subscribe is a useful communication model, which uses the abstraction of "topics" to define interaction between entities. Doing so decouples the producers from the consumers of data and simplifies application logic. Special nodes called "brokers" host topics and perform message transfer from producers to consumers of each topic. The use of intermediary broker nodes allows the producers and consumers to be decoupled from each other. Data producers can send messages to a topic without waiting for it to be received by consumers, and consumers are notified asynchronously for each message. Publish-subscribe systems also maintain a persistent log of messages for each topic, which allows them to ensure strong data-delivery semantics, such as atleast-once delivery of messages to each consumer.

2.5.3 Key-Value Storage

The processing logic of our target applications depend on their state, hence it should be stored in a way that facilitates easy and efficient access. Key-value stores offer a convenient data model, wherein each data-item can be referenced using a unique key for reading and writing. Typically, key-value stores maintain multiple replicas of each data-item to ensure tolerance from failures. The network connectivity of data replicas and the number of replicas chosen for performing a read or write operation determines the operation latency and the consistency.

2.6 Timing considerations for situation-awareness applications

2.6.1 Why cloud-based solutions fall short?

A number of platform services offering compute orchestration (e.g., Kubernetes), publish-subscribe communication (e.g., Apache Pulsar) and key-value storage (Apache Cassandra) are available for the cloud computing ecosystem. These services are widely used since they provide strong data-plane semantics (such as Cassandra's tunable consistency levels and Pulsar's at-least-once message delivery guarantee). In addition to useful semantics, the data-plane of these systems have been tuned for providing high throughput and low latency.

However, relying on cloud-based solutions necessitates communicating with a remote datacenter location through the WAN, and sending all of the data from client's sensors to the datacenter. Traversal through the WAN incurs high and unpredictable communication latency, and the large volume of high-fidelity sensor data (e.g., stereo cameras, LiDAR, etc.) causes high backhaul bandwidth consumption. High latency in the sense-process-actuate control loop of the applications results in their functionality being impaired. On the other hand, high backhaul bandwidth consumption limits the scale at which a particular application can be deployed.

2.6.2 Need to move to the network edge

Edge computing [1] presents a viable deployment alternative for the aforementioned platform services. The presence of computation and storage resources in proximity to the clients make it possible reduce the network latency between the clients and application components. Edge infrastructure is a continuum of geo-distributed *sites* hosted by multiple providers, such as telecommunication network providers, co-location providers (e.g., Vapor IO [12]), etc. An edge site typically comprises of a rack of server-grade machines, equipped with storage and networking infrastructure. Based on the dataset [13] released by Alibaba Edge Node Service, which is the only publicly available dataset about a real-world Edge infrastructure deployment, Table 2.1 shows the size of each Edge site. Edge sites are much more resource-constrained than a typical datacenter, because of space and power limitations.

Table 2.1: Distribution of Edge site capacity in the Alibaba Edge Node Service dataset.

	Minimum	Maximum	Median	Average
CPU cores	96	4337	1158	1367
Memory (GB)	240	20745	4766	5116
Server Count	2	45	14	15

Because of the smaller resource footprint, infrastructure providers deploy a large number of geo-distributed Edge sites so that a large number of users can be supported. However, the geo-distributed nature of Edge site deployment implies that the network connectivity of Edge sites to the Internet is heterogeneous, meaning that they connect to the Internet through different peering points. Due to such a heterogeneity, the network latency from a client varies significantly across different Edge sites. This behavior is also significantly different from the Cloud, where the latency between multiple nodes is almost negligible because the nodes are hosted within the same datacenter.

2.6.3 New mechanisms are needed at the edge

The cloud-based platform services do not offer the same performance when deployed as-is on edge infrastructure because their control-plane policies are not optimized for an edge setting. These systems have been designed for operating in a datacenter setting, wherein machines are connected to each other via a high-throughput low-latency network. Clients, compute and data entities are co-located in the same datacenter, making the network latencies between these components negligible as compared to the compute and data access latencies. Hence, in these systems, the key to ensuring bounded end-to-end latency is uniform load balancing that prevents the formation of workload hotspots and therefore latency inflation. Using such systems as-is in an edge computing environment would result in the placement of compute and data entities on edge sites in a way that is agnostic to network latencies among clients and edge sites - making the satisfaction of end-to-end latency requirements difficult. Furthermore, cloud-based platform services do not monitor observed end-to-end latencies to detect a violation and trigger a reconfiguration to alleviate the violation.

In order to better serve the target applications, we need to introduce new edge-specific mechanisms into the control-plane of the platform services. Doing so will allow them to operate effectively in an edge setting and meet the requirements of applications.

CHAPTER 3
FUNDAMENTAL MECHANISMS FOR GEO-DISTRIBUTED OPERATION OF PLATFORM SERVICES

The challenges imposed by the peculiar characteristics of a geo-distributed Edge infrastructure and situation-awareness applications on the design of control plane policies for platform services necessitate the introduction of novel mechanisms to address them. This dissertation proposes three key mechanisms to address these challenges - Dynamic Spatial Context Management, Network Proximity Estimation and End-to-End Application Monitoring.

The Dynamic Spatial Context Management mechanism allows the control plane of platform services to maintain a frequently updated view of the spatial context of system entities such as application instances, data-items and end-clients. This spatial context information is used to make control plane policy decisions, such as mapping a client to an application instance and determining the set of clients that need to share data for inter-client coordination. The objectives of these control plane policies is to ensure that the spatial affinity requirement of an application is fulfilled, as described in Section 2.3.4.

While the Dynamic Spatial Context Management mechanism is used to establish logical connectivity between system entities based on spatial proximity, it is not responsible for mapping those system entities on to the physical infrastructure - a problem that requires taking into account the heterogeneity in infrastructure topology and latency requirements of applications. The Network Proximity Estimation mechanism allows control plane policies to estimate the network latency between physical nodes in the infrastructure that can be used for the placement of system entities such that applications' latency requirements are satisfied.

Finally, the continuous execution of applications requires the control planes of platform services to monitor the end-to-end latency experienced by each application instance, that includes queuing and computation delays as well as communication delays between different entities. The End-to-End Monitoring mechanisms allows the control plane policies to obtain an aggregated view of the various component latencies making up the end-to-end observed latency so that a violation of the application's requirements can be detected. The end-to-end view of observed latencies also enables root-cause analysis techniques to identify the source of the performance violation and trigger the appropriate reconfiguration action to alleviate it.

In this chapter, we will discuss the three mechanisms proposed in this dissertation in detail. We first present the infrastructure and workload scenario that has been considered to motivate the problems solved by these mechanisms and to design the experimental settings. Then, for each mechanism, we will first describe the objective of the mechanism, enumerate examples of control plane policies that will benefit from it, present the abstractions exposed to the control plane policies, and why previous approaches at attaining this objective are not sufficient. Finally, we demonstrate the use of these abstractions for building useful control plane policies that can support the efficient operation of platform services on a realistic Edge infrastructure against a workload of situation-awareness applications.

3.1 Infrastructure Topology and Client Workload considered

To make the case for the mechanisms proposed in this dissertation, we utilize the dataset [13] characterizing the Edge cloud service of Alibaba Cloud, both at the infrastructure and workload level. At the infrastructure level, it provides detailed information about the number of Edge sites in each city of China and the network provider owning them, the number and size of physical machines in each Edge site and the network RTT between sites. At the workload level, the dataset contains information about the number and size of Virtual Machines (VMs) hosted by each physical machine at different sites and the CPU utilization of each VM. For evaluations in this chapter, we choose the city of Shanghai as the one offering situation-awareness applications to its residents (which is just for clarity of exposition as the evaluation methodology can easily be extended to include application clients in other cities). We simulate client activity (including mobility) in the city of Shanghai ourselves, since the Edge dataset [13] does not provide client activity information. The network connectivity between a client and an Edge site is determined by the location of the client and the geographical areas covered by each Edge site in Shanghai. To estimate the geographical coverage of each Edge site, the precise geographical location of Edge sites and their connectivity with cell towers are needed. However, the dataset only provides a city-level granularity of Edge site locations. Hence, to estimate their precise geo-locations, we gather the locations of cell towers owned by the different network providers from CellMapper [14], perform k-means clustering on them and obtain the likely locations of the Edge sites. The number of k-means clusters for each network provider's cell tower clustering is made equal to the number of Edge sites owned by that provider in Shanghai (from the dataset). Upon obtaining the locations of Edge sites using clustering,

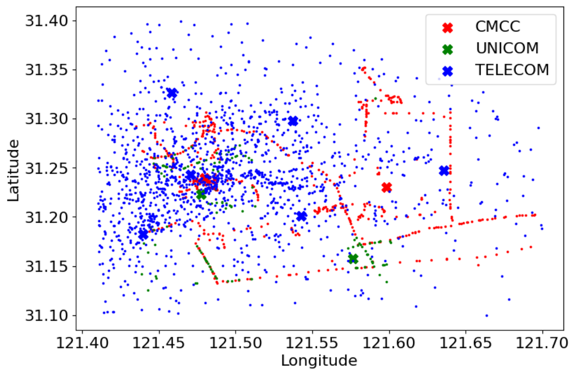


Figure 3.1: Edge infrastructure of the city of Shanghai that is under consideration in this chapter. The crosses denote Edge site locations, while the dots denote cell tower locations. The different colors represent the three telecom providers in Shanghai, namely CMCC, China Telecom and Unicom.

we need to assign resource capacity to each site for which we again use the Alibaba Edge Node Service dataset, specifically the information about resource capacity of each Edge site. To map the resource capacity from the dataset to an Edge site location extracted via clustering, we assign the most resource-rich site's resource capacity in the dataset to the Edge site location that has the maximum number of cell towers in its cluster, and so on. Fig. 3.1 illustrates the infrastructure topology, with the locations of cell towers and Edge sites marked. In addition to Edge sites within Shanghai with precise locations, the physical infrastructure considered in our evaluations also consists of Edge sites in other cities of China. Each Edge site outside Shanghai is significantly farther away from any client considered in the workload (compared to sites within Shanghai), and therefore the city-level coarse-grained location provided by the dataset works for such a site.

3.2 Dynamic Spatial Management Mechanism

Situation-awareness applications, such as collaborative autonomous driving and UAV swarm navigation, interact with the physical environment, by sensing data and performing actions on it. Typically, there are many clients of the same application operating in a common geographical space. In such a setting, a client's processing logic can benefit by incorporating information extracted from other clients' sensed data. For a given client, the set of other clients from which it needs sensor data depends on the location of clients, the size of their sensor range and the area of interest of the given client. The AoI of a client is defined as the geographical region whose data it is interested in receiving. The sensing range of a client depends on the sensor hardware used, e.g., LiDAR, camera, etc. The size of the

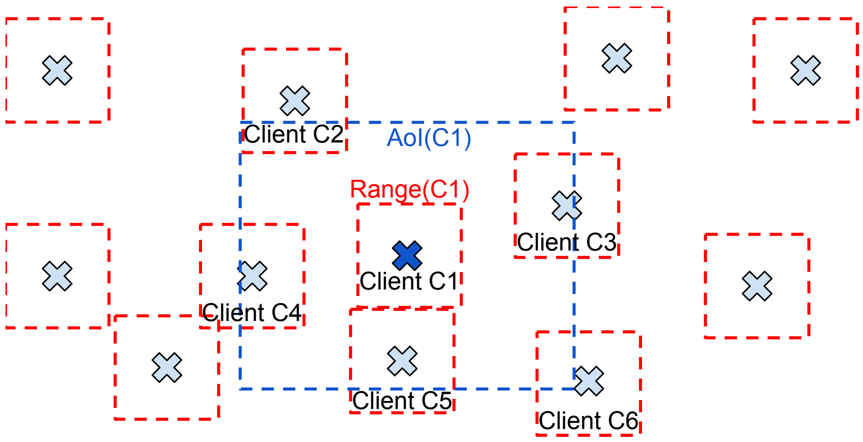
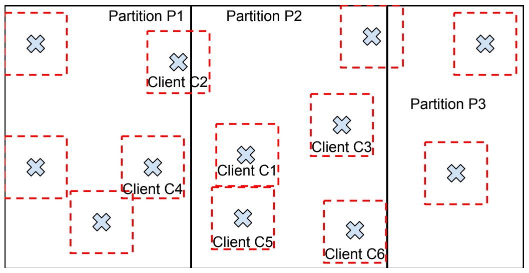


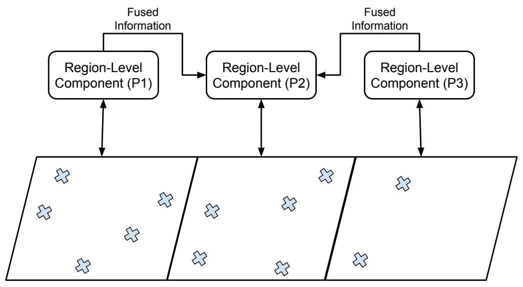
Figure 3.2: An exemplary spatial distribution of situation-awareness application clients overlaid on two-dimensional space. The sensing range of clients is shown as a red dashed rectangle around the client's location. In addition, the Area-of-Interest of client C1 is shown.

Area-of-Interest depends on the nature of the application. For instance, since vehicles in a city are not expected to move at speeds higher than, say, 25 miles per hour (in urban cities in USA), the size of AoI of vehicular clients for the collaborative perception application is bounded (e.g., 150 meters x 150 meters in the TalkyCars system [6]). There exists only a finite region around a given vehicle that would contain any interesting event for that vehicle. Fig. 3.2 shows a typical arrangement of clients in geographical space with the sensing range of each client marked in red and the AoI of one of the clients (C1) marked in blue. Each client senses data from the environment within its range and the application instance serving it generates actionable information from the sensed data. The application instance serving client C1 is interested in receiving all actionable information about the geographical region within the bounding-box of C1's AoI. Hence, the application instance serving C1 needs to receive processed information from all other application instances that are serving those clients whose range overlaps with the AoI of C1.

An intuitive way of modeling these applications was discussed earlier in Section 2.3.1, wherein a region-level component is responsible for fusing the information extracted from multiple clients to realize inter-client coordination. However, in a large-scale geo-distributed deployment of such applications, a single region-level component would be insufficient to serve all the clients - because of scalability limitations in its implementation, resource constraints on the Edge site hosting it, or both [6]. Hence, to support a large number of clients, multiple instances of the region-level component are maintained as shown in Fig. 3.3, with each instance serving a distinct partition of the entire geographical area. All clients within



(a) An example partitioning of geographical space. Each partition is to be managed by a distinct application component. The important point to note is that each partition is not self-sufficient, but rather relies on information from other partitions as well. The data-dependence between partitions results from the fact that the AoI of clients inside a given partition overlaps with the range of clients in other partitions.


Fused Information
Fused Information
Region-Level Component (P1)
Region-Level Component (P2)
Region-Level Component (P3)

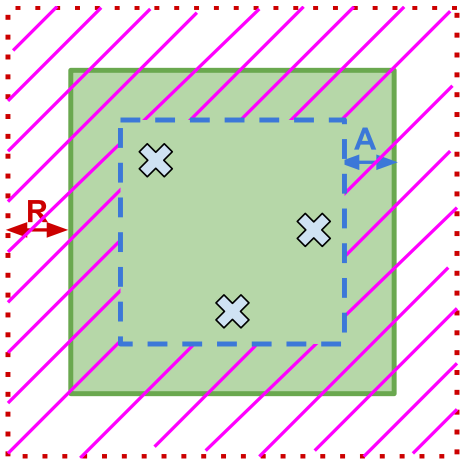
(b) Clients in each spatial partition are mapped to a unique region-level component that is responsible for fusing the information extracted from each individual client.

Figure 3.3: Illustration of partitioning of geographical space to support large-scale deployment of situation-awareness applications that require coordination among clients.

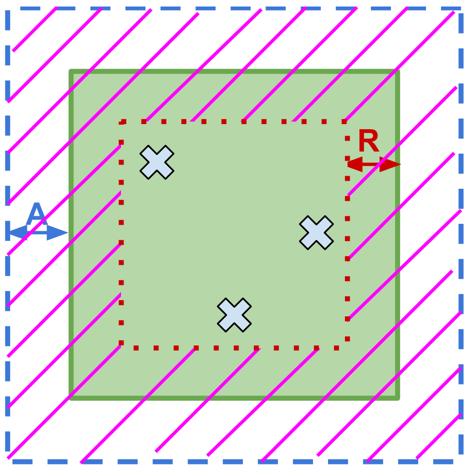
a given spatial partition should be served by the same region-level instance, thus enabling inter-client coordination among all clients within that partition. Due to the inherent mobility of clients and the fact that the spatial partitioning is not necessarily aligned with the sensing range of each client, a client's AoI can overlap with another client's range that is located in a different partition. For example, in Fig. 3.3a client C1's AoI overlaps with the range of client C4, however C1 and C4 belong to partitions P2 and P1 respectively. To make sure that information from client C4 is taken into account while processing the sensor data of client C1, the region-level component instance serving client C1 receives a stream of fused information from the instance serving client C4, as shown in Fig. 3.3b.

For situation-awareness applications that require inter-client coordination, mapping clients to region-level component instances needs to be done by taking into account the spatial context of clients (denoted by their location) and that of the region-level application component instances (denoted by spatial partition they are meant to serve). Information sharing between region-level instances is also dictated by the spatial context of the different instances and the range and AoI of clients served by those instances, as shown in Fig. 3.4.

Thus, to ensure that each client is served by an application instance specific to its spatial context and that application instances are able to share relevant data among each other, it is



(a) An illustration of the information generated by a given region-level component instance that needs to be shared with other instances.



(b) An illustration of the information needed by a region-level application component from other application instances.

Figure 3.4: The shaded regions in both images show the data that needs to be shared with other region-level instances (Section 3.2) and that needs to be received from other instances (Fig. 3.4b). The R denotes the size of sensing range, while A represents the size of AoI of each client. The shaded area represents the maximum extent of information that needs to be shared/received for any possible client location. In Section 3.2, it is assumed that the recipient clients are located just outside the green region, while sending clients are located just inside the green region. Similarly, in Fig. 3.4b, it is assumed that the recipient clients within the green region are present right at the boundary, while the source clients are located just outside of the green region.

imperative to treat the spatial context of clients, application components and data-items as first-class attributes.

The main challenge in doing so is to handle continuous client mobility, that results in the AoI and spatial context of each client to change continuously. Hence, a static mapping of clients to application instances would result in frequent violations of the spatial affinity requirement, wherein a client would be communicating with application instances that serve a different spatial region than the one in which the client is located. Similarly, the set of data-items that a client's application instance is interested in also changes along with client mobility. Furthermore, given the continuous mobility of clients, workload skews are common - caused due to a large number of clients accumulating in a particular spatial partition, e.g., in the event of a surge in vehicular traffic in a city's downtown area due to a concert. Due to limited resource capacity of edge resources, such a workload skew can result in a performance degradation on the application instance serving that spatial partition. Therefore, spatial workload skews necessitate the monitoring and remapping of geographical regions to applications instances and data-items, so that the skews can be minimized and performance degradations can be avoided.

3.2.1 Control plane actions that need this mechanism

There are two main actions taken by the control planes of platform services that require spatial context information of clients and compute and data components:

Dynamic Client-to-Application Mapping. Situation-awareness applications require that a client be connected to a region-level application component instance that is assigned to the spatial partition within which the client is currently located. A metric that quantifies the goodness of this mapping is Spatial Alignment, that measures how many of the expected clients that should have been mapped to an application component are actually mapped. The Spatial Alignment metric for spatial partition A is quantified in Eq. (3.1).

$$SA(A) = \frac{\text{max. clients in } A \text{ served by the same app instance}}{\text{number of clients in } A} \tag{3.1}$$

The control plane policy for client-to-application mapping would ideally ensure the spatial alignment metric for all spatial partitions to be 1.0, meaning that all clients that are currently located in a given spatial partition A are mapped to the same application instance.

Area-of-Interest Queries. Clients and application components need to query data-items whose spatial context overlaps with the querying entity's AoI. This query is served by the control plane that evaluates the spatial context overlap with the querying entity's AoI and returns a list of satisfying data-items. We use a metric called AoI Satisfaction Rate to quantify the goodness of the query result, as shown in Eq. (3.2).

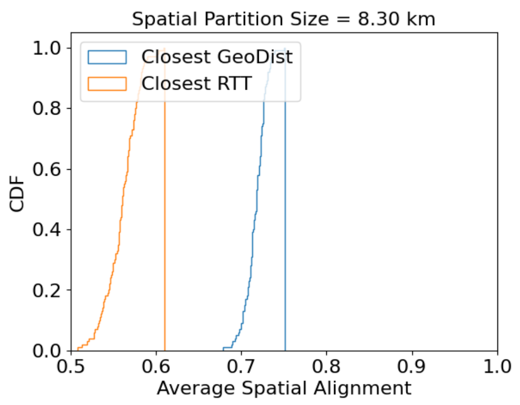
$$\text{AoI Satisfaction Rate} = \frac{|\{e : e \text{ is returned by query and } e \in AoI\}|}{|\{e : e \in AoI\}|} \tag{3.2}$$

Ideally the control plane policy for evaluating these queries should be able to achieve an AoI Satisfaction Rate of 1.0.

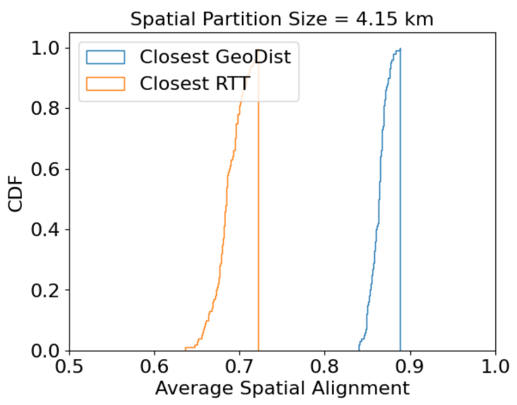
3.2.2 Limitations of previous work in implementing policies

Contemporary Edge computing solutions do not allow coordination among application components that are deployed across multiple Edge sites [15, 16]. Hence, a client c that is mapped to an Edge site E can only coordinate with other clients that are also mapped to the site E. Similarly, c can only discover other system entities (data-items and application instances) on the site E. We consider 2 high-level approaches in previous work to map clients to Edge sites - mapping each client to the geographically closest site (GeoDist) (as in Lähderanta et al. [17]) and to the site with smallest RTT from the client (RTT) (as in Saurez et al. [18]). In the following evaluations, we show how both these approaches are deficient in satisfying the spatial alignment requirement of client-to-application mapping as well as serving Area-of-Interest based queries.

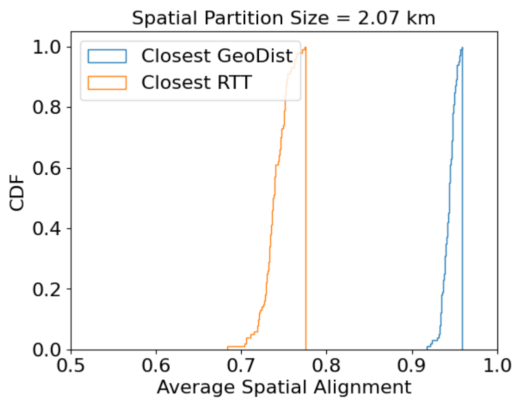
Spatial Alignment Evaluation. To evaluate the above client-to-application mapping baseline heuristics in terms of spatial alignment, we first consider the area of the city under evaluation and divide it into a number of spatial partitions, within which we ideally expect clients to be mapped to the same application instance (thereby creating a perfect spatial alignment). The size of spatial partitions is varied in the experiment to represent a diverse set of applications. We create 1000 clients (with equal number of clients connected to each network provider) and place each one of them at a cell tower location. The placement of clients at cell tower locations is justified by the fact that the spatial distribution of cell towers follows that of client activity. This client placement is randomized and the experiment is repeated 100 times. For each experiment run, we compute the average spatial alignment over all spatial partitions in the scenario. Fig. 3.5 shows the distribution of the average spatial alignment that results from mapping clients to application instances using greedy heuristics that aim at minimizing geographical distance and network RTT between the client and Edge site. The metric shown is average spatial alignment, that is the average



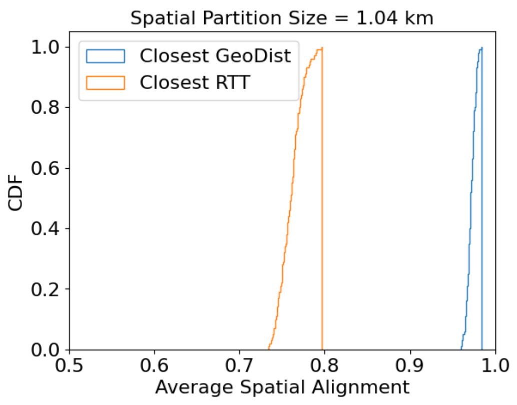
(a)



(b)

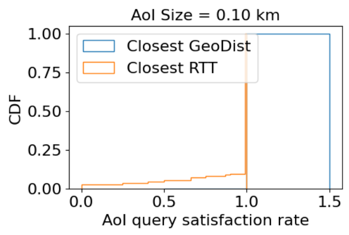


(c)

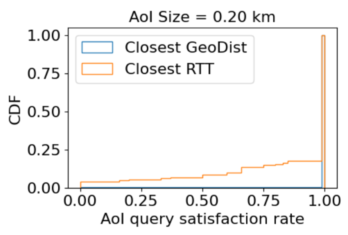


(d)

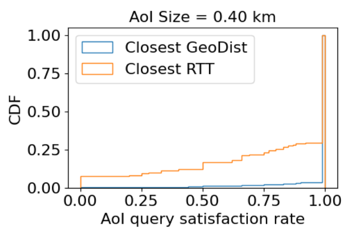
Figure 3.5: Distribution of Average Spatial Alignment observed for different sizes of each spatial partition. Closest RTT based mapping of clients to Edge sites results in worse spatial alignment compared to Closest Geo Distance based mapping.



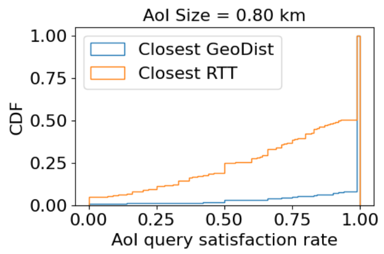
(a)



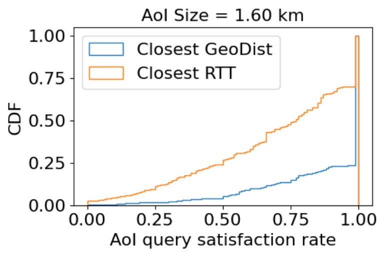
(b)



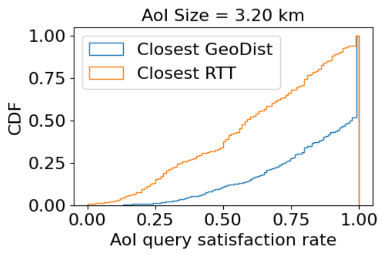
(c)



(d)



(e)



(f)

Figure 3.6: Distribution of the AoI Satisfaction Rate achieved when using two baseline client-to-site mapping approaches - Closest GeoDist and Closest RTT. Both approaches fail to achieve perfect AoI Satisfaction Rate especially at higher AoI Sizes.

of the spatial alignment of all the spatial partitions in that experiment run. Although the baseline policy GeoDist, that selects the geographically closest Edge site, is able to attain a high enough average spatial alignment, it does not attain the perfect 1.0 value because not all clients in a given spatial partition always have the same Edge site as the closest. Furthermore, the Closest RTT approach offers even worse spatial alignment because often within a given spatial partition, clients that belong to two different network providers may be bound to different Edge sites.

AoI Satisfaction Evaluation. Next, Fig. 3.6 shows the distribution of AoI query satisfaction rates offered by the Closest GeoDist and Closest RTT baseline policies for retrieving the data-items with spatial context overlapping with the AoI of the querying client. For this evaluation, we consider the scenario wherein each client is associated with one data stream, that contains the sensor data collected by the client. Each client has a dedicated application instance that processes its sensor data, while also consuming the data stream of other clients in its AoI. We create one client at every cell tower location, and each client's application instance submits an AoI query to find which other clients are in its AoI. Depending on the client-to-site mapping, a variable number of AoI members are returned, that is compared with the ground-truth set of clients within the AoI to compute the query satisfaction rate. The size of AoI query is varied to represent a diverse set of applications and their satisfaction rate distributions are shown. Both client-to-site mapping approaches perform poorly and fall significantly short of the expected AoI Query Satisfaction Rate of 1.

Both of the above approaches fall short in terms of providing high Spatial Alignment for client-to-application mapping and AoI Query Satisfaction Rate for serving AoI-based queries. The evaluated approaches fall short because they do not incorporate the spatial context of application instance and data-items nor the AoI of clients. Instead they rely solely on the client-to-Edge-site mapping, wherein the Closest GeoDist approach is based on the locations of clients and Edge sites, while the Closest RTT approach is based on the network infrastructure topology serving clients and Edge sites. Therefore, we need a new mechanism that takes into account the spatial context and AoI specified by the application to perform client-to-application mapping and serving range queries. The mechanism should be able to associate unique spatial contexts to application instances and map clients to these instances based on client location. It should also maintain a spatial index of data-items and filter out the result of an AoI-based query by determining which data-items fall within the AoI.

3.2.3 A New Mechanism for Spatial Context Management

The Dynamic Spatial Context Management mechanism is utilized by both the control plane's compute and data placement policy of platform services as well as the client library of the platform service running on the client application. The control policy interacts with this mechanism to maintain the spatial context of the system entities (data and compute) while the client library uses this mechanism to keep track of the spatial context of clients. Both these types of spatial contexts are useful in determining which compute/data entities a given client should access.

Interface exposed to the control plane

The mechanism divides the geographical space into rectangular regions called *Tiles*, that are of arbitrary size. We choose rectangular tiles so that they can best approximate the spatial partitions expected by collaborative situation-awareness applications. An example partitioning is shown in Fig. 3.3a. Each Tile has a unique Tile ID, that can be used to directly reference the Tile. Each tile contains a number of entities, such as clients or data-items within it. In the following list, we enumerate the various functions that the mechanism exposes to the control plane for accessing and updating the spatial partitioning.

- **GetTileID**. This function is used to convert a location to the ID of the tile that it belongs to. The function is used by the control plane policy to perform client-to-application mapping.

Input: Location of the client

Returns: ID of the tile within which the client location exists.

- **GetTileCoverage**. This function determines the bounding box of the spatial area covered by a given tile. This function is used by the control plane policy to perform client-to-application mapping as well as by the libraries on the client and application components to detect if a change of mapping is needed in the event of significant client mobility.

  Input: ID of the tile.

  Returns: Bounding box that represents the spatial area covered by the tile.

- **UpdateEntityLocation**. This function is used to update the location of an entity, that could be a client or a data-item. In addition, if the new location of the entity is in a different tile than its previous location, this function would move the entity from previous tile to the new tile.

  Input: ID of the entity and its current location.

- **GetIntersectingTiles**. This function is used by the client and application component library to perform range queries to find out which entities are within a given geographical range.

  Input: Bounding box of the area within which to query for intersecting tiles.

  Returns: A set of tiles that intersect with the provided bounding box.

- **SplitTile**. This function call is used by the control plane to split a tile into two. The split is carried out in a way to ensure that the number of entities in the two resultant tiles is (almost) equal. This function is used to alleviate load on a tile that is going through a workload surge.

  Input: ID of the tile to be split.

  Returns: IDs of the children tiles created by splitting input tile.

- **MergeTiles**. This function allows the control plane to trigger a merging of two adjacent tiles. This function is called when the tiles being merged are experiencing workload under-utilization, and their total workload can be handled by one tile along.

Input: IDs of the tiles to merge.

Returns: ID of the tile created as a result of merging input tiles.

Demonstration of using the mechanism for implementing a control plane policy

We now demonstrate how the proposed mechanism and the associated API are useful in implementing control plane policies for platform services. For this mechanism, we use an application orchestrator as the driving example. Algorithm 1 shows the pseudocode of the control plane policy for finding a suitable application instance for mapping a client based on its geographical location and the spatial contexts of various region-level application instances. The control plane maintains a mapping $APPS$ between an application instance and the Tile that represents the geographical area served by the application instance. The policy first determines the Tile in which the client is currently located. If the client's tile already has an application instance associated with it, the policy returns connection information about that instance. Otherwise the policy deploys a new application instance for the client's tile, maps the application instance to the tile and returns its information to the client. The policy also adds the client to the tile to update the occupancy information, that can be used to detect overload and trigger re-partitioning.

Algorithm 1 Handling Deploy Request from Client

```
Require: client $c$
Require: client's location $loc$
  $t \leftarrow GetTileID\left(loc\right)$
  if $APPS\left[t\right]\ exists$ then
      $A \leftarrow APPS\left[t\right]$
  else
      Deploy app component $A$ for tile $t$
      $APPS\left[t\right] \leftarrow A$
  $UpdateEntityLocation\left(t, c\right)$
  $SendConnectionInfo\left(c, A\right)$      ▷ Send info to client for connecting to app component
```

Algorithm 2 describes the control plane policy for handling an overloaded region-level component due to workload skew. This policy is triggered by the control plane when it identifies that a given application instance is overloaded by the compute requirement of serving all the clients that are currently present in the tile its is mapped to. Hence, the tile needs to be split and workload divided among two application instances. The policy takes as input the tile associated with the overloaded application instance. The policy uses the *SplitTile* function to create two new tiles. It reuses the application instance for the old tile for one of the new ones, and deploys another instance for the second new tile. Each client that belonged to the old tile is now a part of one of the two new tiles. The policy sends the connectivity information of the new application instances to the respective clients. Although not shown, the *MergeTiles* functionality provided by the mechanism can be similarly used to merge two tiles together if the two application instances serving these two tiles are sufficiently underutilized.

Algorithm 2 Handling an Overloaded Tile

```
Require: overloaded tile $t$
  $t1, t2 \leftarrow SplitTile\,(t)$
  $APPS\,[t1] \leftarrow APPS\,[t]$
  Deploy app component $A$ for tile $t$
  $APPS\,[t2] \leftarrow A$
  for client $c \in GetEntities\,(t1)$ do
      $SendConnectionInfo\,(c, APPS\,[t1])$
  for client $c \in GetEntities\,(t2)$ do
      $SendConnectionInfo\,(c, APPS\,[t2])$
```

Improvement over Previous Approaches

In Section 3.2.2, we have shown how the previous work in mapping clients to application instances and serving Area-of-Interest-based queries fall short in terms of meeting the spatial affinity requirements of situation-awareness applications. Here we briefly show the benefit of using a mechanism for maintaining client-to-application mapping based on the spatial context of application instances and location of clients. The details of the mechanism are presented in Section 4.1. We simulate the mobility of 200 clients and use the spatial context management mechanism to map clients to application instances. We measure the spatial alignment over time and plot it in Fig. 3.7. The spatial alignment offered by the proposed mechanism remains close to 1.0 for most of the time, with the exception of some dips which are caused either due to dynamic updates to the spatial partitioning by the mechanism or due to the control plane taking some time to update the client-to-application mapping when clients move to different spatial partitions. Fig. 3.8 shows the behavior of spatial alignment metric more clearly, wherein we can see that for the vast majority of time instants the spatial alignment value is a perfect 1.0. Therefore, the use of a mechanism that explicitly maintains the spatial context of application instances and uses the geographical location of clients to map them to application instances can satisfy the spatial affinity requirements of applications.

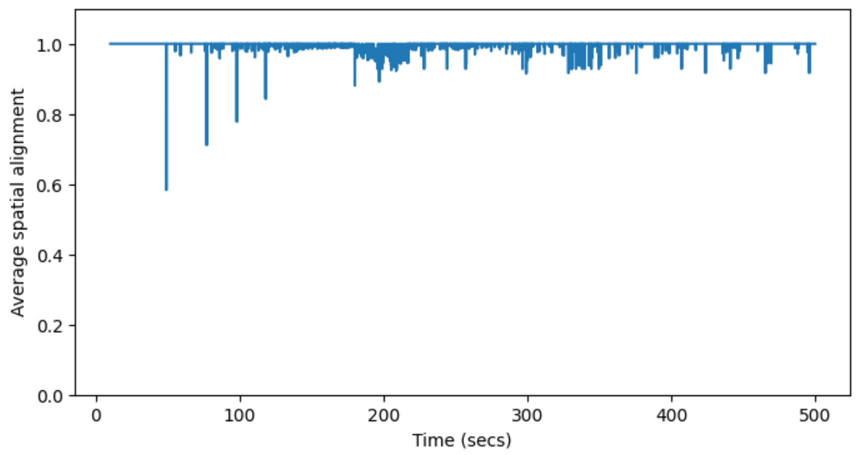


Figure 3.7: Spatial Alignment over time. The dips in spatial alignment are due to updates to the spatial partitioning as well as clients moving from one spatial partition to the other and the control plane taking some time to update the client-to-application mapping.

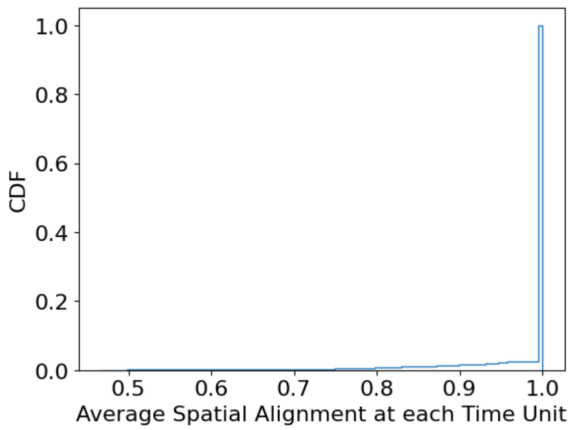


Figure 3.8: CDF of Spatial Alignment values over all time instants. Each data point in the CDF corresponds to a point in Fig. 3.7.

3.3 Network Proximity Estimation

The critical path of applications' control-loop contains system entities (e.g., clients and application components) communicating with one another and accessing data-items. In a densely geo-distributed and heterogeneous infrastructure such as the Edge, the network latency between a client and an Edge site varies significantly depending on which site the client is communicating with. Similarly, the network latency between different Edge sites is also heterogeneous (as discussed in Section 3.3.2). Hence the choice of Edge site for hosting compute or data components of an application instance has a significant effect on the perceived end-to-end latency of the application, be it the sense-process-actuate control-loop's latency or the latency of accessing data from other clients for inter-client coordination [19, 20, 21, 22]. Therefore, placement policies in the control plane of platform services need to be aware of the topology of the underlying infrastructure to make decisions. For instance, in the case of the collaborative perception application, the placement of application components should be such that the end-to-end processing latency is bounded under the application's threshold. Similarly, in the case of the drone swarm coordination application, the publish-subscribe topic should be placed on a suitable broker node to ensure that the end-to-end message delivery latency is bounded under the application's threshold.

Therefore, the platform services require a mechanism that enables their control plane to estimate the network latency between the end-clients and Edge sites as well as across Edge sites. Such a mechanism would be able to estimate the network latency between a pair of entities quickly and with low overhead, making it suitable to be used as a part of the placement policies of these platforms that evaluate a number of Edge sites as candidates for compute or data placement. Given the dynamic nature of the Edge infrastructure and client mobility, the network proximity estimation should also be able to adapt to dynamic changes in network latency between system entities.

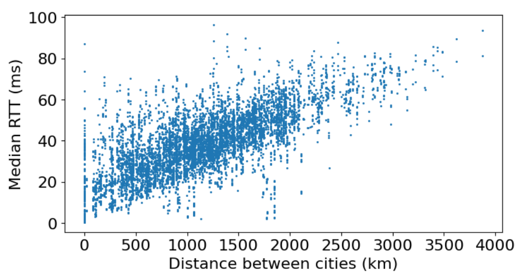
3.3.1 Control plane policies that need this mechanism

Network proximity estimation is useful for control plane policies that map logical system entities, such as application instances or publish-subscribe topics, to physical nodes in a way that the end-to-end latency requirements of applications are met. Previous works in this space [20, 21, 22] rely on inter-node communication latency estimates for making these decisions. However they do not go into detail about how these estimates are obtained, assuming instead that they are readily available for use by the policy. In the case of the application orchestrator platform service, that manages situation-awareness applications that consist of a pipeline of multiple components [23, 24], the end-to-end latency is the sum of the processing latency at each application component and the communication latency between each upstream-downstream pair of components. Similarly, in the case of a publish-subscribe system, the end-to-end messaging latency for a given topic is the sum of the communication latency from the publisher clients to the broker, the processing latency on the broker and the communication latency from the broker to the subscriber clients. In the above two systems, estimating the processing latency of application components and publish-subscribe broker can be done using previous works in the cloud computing realm as well [25]. However, estimating the communication latency between a pair of nodes requires rethinking and potentially coming up with a new mechanism. Both the resource allocation policy for the application orchestrator and topic placement policy for publish-subscribe system would benefit from such a new mechanism.

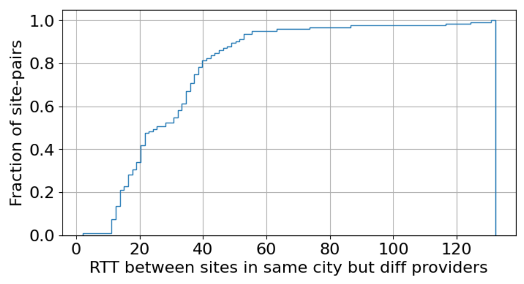
3.3.2 Limitations of previous work in proximity-aware compute/data placement

Control plane policies for ensuring that compute and data entities accessed by a given set of clients is placed in their proximity have been one of the main directions of previous research in Edge computing. The use of geographical distance as a proxy for network latency has been commonly used, given the simplicity of the scheduling logic once the geo-locations of system entities (e.g., clients and edge sites) are known. Sarkar and Misra [19] propose the transformation of geographical distance between a client and candidate Edge site into the network latency between them using a linear transformation $rtt\,(ms) = dist\,(km){*}0.02{+}5$ [26]. Other works do not rely on the use of such a transformation, but rather perform greedy placement on the geographically closest site to ensure low-latency access to the data or compute entity [17]. Geographical location has also been used in a more coarse-grained manner, in that the placement of compute/data entities is specified to be within a large location, e.g., within a city [27]. The selection of the specific Edge site is done on the basis of a second order policy logic that aims to ensure even load distribution among the various Edge sites in that particular coarse-grained geographical area.

The above approaches fail to work in realistic edge settings because they assume that geographical proximity is correlated with network proximity, which is not true because of lack of uniformity in the way in which different network providers are peered with each other. Two systems entities (a client and an Edge site, or two Edge sites) in close physical proximity might have to communicate through extended routing paths because



(a) Variation of network RTT with geographical distance between Edge Sites.



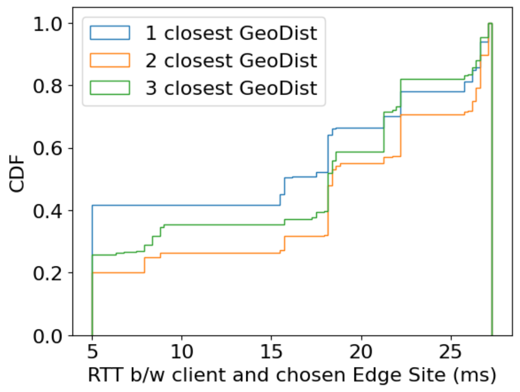
(b) Client-Site RTT for sites selected by transforming the geographical distance into network RTT and uniformly choosing one among the set of sites satisfying the latency constraint.

Figure 3.9: Variation of network RTT between edge sites with respect to geographical distance.

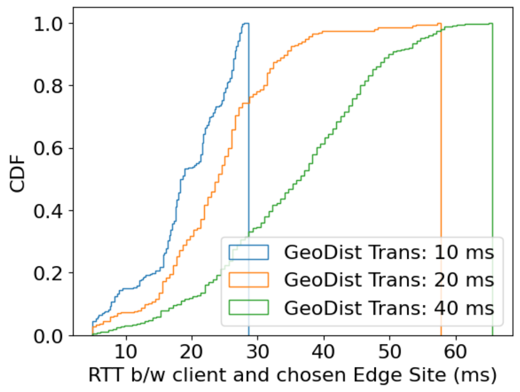
of the peering between the network providers serving the two entities. In Fig. 3.9a, we present the variation of network round-trip times between an Edge site located in Shanghai to other Edge sites throughout China. It shows that although network latencies are loosely correlated with geographical distance, there is a significant amount of variance, that will result in placement policies choosing an incorrect Edge site for hosting a compute or data entity. In Fig. 3.9b, we specifically measure the network round-trip times between sites that are located in the same city, but are served by different network providers. The high round-trip times implies that the high variation in Fig. 3.9a is due to the expensive peering between different network providers hosting Edge sites.

We evaluate the efficacy of the aforementioned baseline proximity estimation techniques - the greedy distance-minimization approach and the one that transforms geographical distance to network RTT. For this experiment, we consider the infrastructure topology of the city of Shanghai as described in Section 3.1. We simulate clients at randomly (uniformly) chosen cell tower locations and aim to find an Edge site to host its application instance using one of the above two policies. The goodness of an Edge site selection is quantified by the observed RTT between the client and chosen Edge site.

The Greedy policy always selects the closest Edge site in terms of geographical distance between the client and the site. In case the chosen Edge site is undergoing compute overload, the policy selects the next closest site, and so on. The other site selection policy that we evaluate is one that transforms the geographical distance between client and Edge site to network round-trip time and uses this estimate to filter out those sites that can meet the latency bound imposed by the application. Among the filtered sites, the policy uniformly



(a) Client-Site RTT for sites selected by Greedy approaches. In the case that the closest site is overloaded, the next closest site is considered and so on.



(b) Client-Site RTT for sites selected by transforming the geographical distance into network RTT and uniformly choosing one among the set of sites satisfying the latency constraint.

Figure 3.10: Performance of Edge Site selection approaches that rely on geographical distance as a proxy for network proximity.

selects one so as to minimize workload skews.

Fig. 3.10a shows the observed RTT when selecting a site using the greedy approach. For more than 50 percent of the clients, the geographically closest site is not the one that it is directly connected to over the network, and hence, traffic needs to go through the network provider's core, or through the Internet to get to it. Since there is no strict correlation between geographical distance and network RTT, this behavior is similar for second and third closest sites as well. For the distance-RTT transformation based approach, we compare the site selection with an application-imposed RTT bound of 10, 20 and 40 milliseconds. We see however, that the transformation does not sufficiently filter out infeasible sites and a large majority of clients have the latency bound violated.

The key takeaway from the above experiments is that both the above approaches that use geographical distance between system components to estimate network proximity fail to fulfill their objective of selecting an Edge site that has low communication latency from the client. The fundamental flaw with them is that they overlooks a key property of real-world network topologies, in that the network route taken by packets seldom aligns with the geographically shortest path between the source and destination entities. Furthermore, there are additional delays caused at the various intermediate hops. Hence, we need a new mechanism for network proximity estimation that derives the proximity information from actual measurement of the network latency between system entities.

3.3.3 A New Mechanism for Network Proximity Estimation

We propose a new mechanism for network proximity estimation that computes the proximity estimation based on actual network round-trip time measurements between system entities.

Interface exposed to the control plane

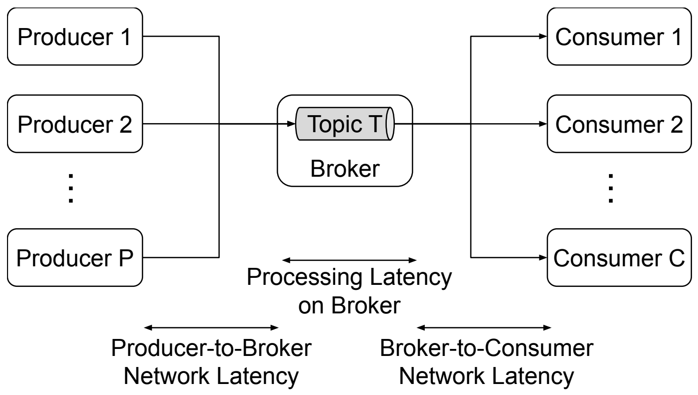
All system components of the platform service (clients, worker nodes, data nodes, etc.) are assigned a unique network proximity identifier. The network proximity mechanism then provides the platform service a function **NetworkRTT** that takes as input the IDs of two system components and returns the estimated network round-trip time between the two components.

Using network proximity mechanism for compute/data placement

We now demonstrate the use of the proposed Network Proximity Estimation mechanism for implementing a control plane policy, specifically the broker selection policy for a publish-subscribe system (Section 3.3.3). This control plane policy selects the broker to host a given topic such that the end-to-end message delivery latency for all publisher-subscriber pairs is under the latency threshold for the topic, as shown in Section 3.3.3. As described in Algorithm 3, it iterates over all the potential candidate brokers and computes the worst-case communication latency if the topic is hosted on each of those brokers. The worst-case communication latency is the sum of the maximum publisher-broker network latency and the maximum broker-subscriber network latency. The sum of worst-case communication latency and processing latency on the broker gives the worst-case end-to-end latency for that topic. All the brokers that have the worst-case end-to-end latency under the topic-specific threshold are selected as candidates. The policy finally selects the candidate broker currently serving the lowest message rate among all candidates to ensure load balancing among brokers.

Improvement over Previous Approaches

We evaluate the performance of the proposed Network Proximity Estimation mechanism by performing an experiment similar to the one described in Section 3.3.2. We place clients randomly at cell tower locations and aim to find Edge sites within different RTT thresholds. In this experiment, the Edge site selection policy utilizes the network proximity between



Producer 1
Producer 2
Producer P
Topic T
Broker
Consumer 1
Consumer 2
Consumer C
Processing Latency on Broker
Producer-to-Broker Network Latency
Broker-to-Consumer Network Latency

Figure 3.11: An instance of publish-subscribe service as an exemplary platform-service that requires information about network communication latency to compute the end-to-end latency experienced by the application. In this example, the end-to-end latency is the message delivery latency from any producer to all consumers for a given topic. The end-to-end latency is given by the sum of the maximum network latency from any producer to the broker, the processing latency on the broker, and the maximum network latency from the broker to any consumer.

Algorithm 3 Broker selection policy for topic T with end-to-end message delivery latency threshold L_{th}

```
Require: topic $T$
Require: latency threshold $L_{th}$
  $prod \leftarrow$ set of producers for $T$
  $cons \leftarrow$ set of consumers for $T$
  $candidates \leftarrow \{\}$
  for each broker $b$ do
      $nw\_lat \leftarrow max_{p \in prod} \frac{1}{2} \cdot NetworkRTT\left(p, b\right) + max_{c \in cons} \frac{1}{2} \cdot NetworkRTT\left(c, b\right)$
      $e2e \leftarrow nw\_lat + proc\_latency\left(b\right)$
      if $e2e \leq L_{th}$ then
          $candidates \leftarrow candidates \cup \{b\}$
  return broker in $candidates$ with lowest $msg\_rate$
```

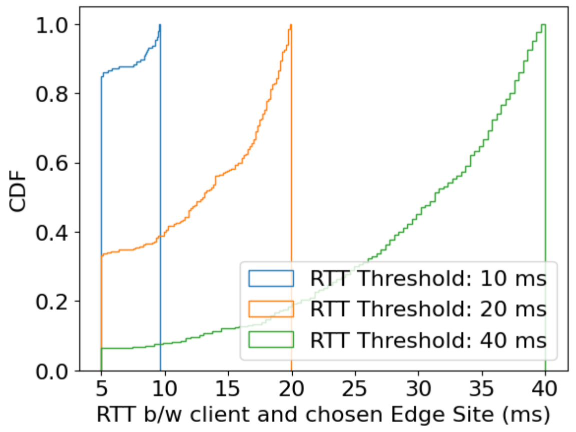


Figure 3.12: Distribution of client-Edge-site RTT for different RTT constraints when site selection is done by using the Network Proximity Estimation mechanism. The proposed mechanism ensures that the selected Edge sites satisfy the RTT constraint.

clients and Edge sites to estimate the network RTT between them and uses it to filter the suitable sites. Fig. 3.12 shows the distribution of the RTTs observed from the Edge site chosen for clients. The Edge site selection policy is able to satisfy all the constraints on RTT and outperform the geographical distance based approaches. The design-space exploration of the network proximity estimation mechanism has been discussed in detail in Section 4.2.

3.4 End-to-End Monitoring Mechanism

Thorough monitoring of applications running atop platform services is a non-trivial task because there can be multiple sources of performance violation. For instance, as shown in Fig. 3.13, the observed end-to-end latency of the application instance is a sum of the queuing, execution and downstream communication latencies of each operator in the pipeline. Thus, detecting a violation of the end-to-end latency requirement and identifying the root-cause requires monitoring measurements of all the component latency metrics as independent streams and their aggregation and analysis. Each platform service has a notion of an *application unit*, that represents an independent set of system entities that do not affect the performance of entities not within the given unit. An example of an application unit is a publish-subscribe topic, that consists of all the producers, consumers and broker hosting that topic. Furthermore, each platform service has its own set of metric streams that need to be monitored for each application unit. The measurements from these metric streams then need to be aggregated and analyzed in way that is specific to the platform service for

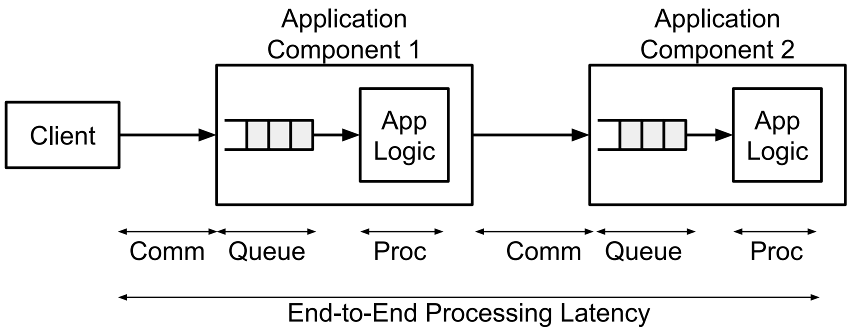


Figure 3.13: A breakdown of the end-to-end processing latency of an application pipeline into its constituent latencies.

detecting violations. The monitoring mechanism needs to scale with an increasing number of application instances being monitored and impose low overhead on the applications and infrastructure.

3.4.1 Control plane policies that need this mechanism

Situation-awareness applications require that the end-to-end latency requirement is satisfied for the entire lifetime of the application, so that correct functionality can be guaranteed. However, given the constant mobility of end-clients and the stringent latency requirements, their requirements are likely to be violated repeatedly and frequently. For instance, a car moving away for the Edge Site currently serving the vehicle-local processing component would incur higher end-to-end processing latency due to higher communication delays. Hence, the control plane of the platform services are required to constantly monitor the running applications for such violations, perform root-cause analysis for determining the cause of the violation, and take appropriate reconfiguration action(s) to resolve the violation. In the above example, a reconfiguration would involve the migration of the application component to an Edge Site that is closer (in terms of network proximity) to the end-client to ensure end-to-end latency satisfaction.

3.4.2 Shortcomings of previous work

Application monitoring has been an active area of work in the cloud computing space with a number of research projects and commercial offerings available. In the context of cloud computing, applications typically do not possess end-to-end latency requirements and the goal of monitoring is to ensure that the tail latency of a given service (that could consist of multiple application components) does not increase significantly. Prior art such

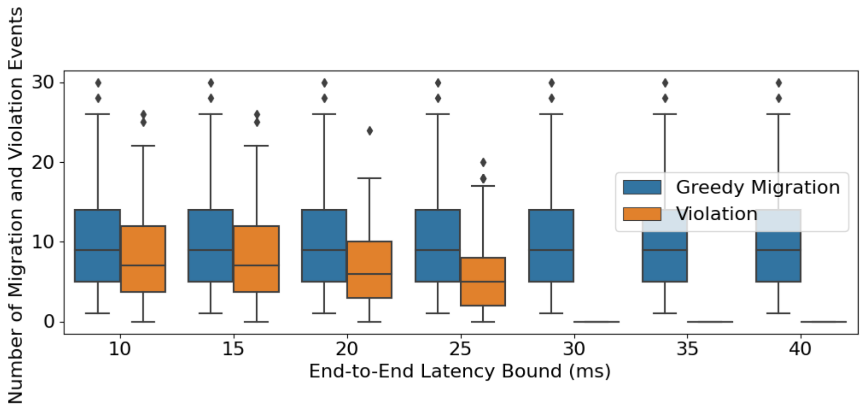


Figure 3.14: Number of migrations triggered by the Greedy monitoring approach along with the actual number violations faced by clients. The end-to-end latency threshold is varied along the x-axis.

as Prometheus [28] and Monasca [29] do not support the aggregation of multiple metric streams to study the behavior of end-to-end latency. Furthermore, their architecture is a fully centralized one, resulting in the use of network bandwidth to send the monitoring data to one specific location (e.g., the Cloud).

Monitoring systems designed for Edge computing environments also suffer from limitations, wherein they aggregate measurements from metric streams over geographical regions instead of aggregating multiple metric streams pertaining to the same application instance [30, 31]. Such systems are not capable of detecting violations of end-to-end latency constraints. Previous contributions to building systems services such as publish-subscribe systems [32] and distributed application runtimes [18] consist of monitoring subsystems that only monitor the the network connectivity between a client and the Edge site that it is connected directly to. Since these solutions do not consider end-to-end latency as the primary metric, they result in triggering more reconfigurations to keep clients connected to the closest Edge site than needed.

We show an evaluation of a greedy monitoring approach that aims to minimize the last-mile network latency between the client and the Edge site it directly connects to. The metric of interest is the number of migrations triggered by such a policy, which should be compared to the number of times that the application's latency threshold is actually violated - where a migration is indeed necessary. We simulate the mobility of 200 independent clients in the city of Shanghai using the Random Waypoint mobility model and track the number of migrations triggered by the greedy policy. In Fig. 3.14, we present the migrations triggered by the greedy policy and the number of latency violations for each client with increasing end-to-end latency bound. We see that as the end-to-end latency bound increases, the number of latency violations decreases, but the number of migrations triggered by the greedy policy remains the same. This behavior is explained by the fact that although the greedy policy triggers the same number of migrations as it is independent of the end-to-end latency constraint, the number of latency violations decreases with an increase in the threshold.

The limitations of an approach that considers only a component of the end-to-end latency necessitates that we come up with a mechanism for monitoring the end-to-end latency by analyzing all the component latencies within it. Violation detection needs to be performed for each application-level unit, or application instance, such as a publish-subscribe topic, or one instance of an application pipeline (as shown in Fig. 3.13). Hence, measurements from metric streams that belong to the same application unit should be aggregated together for violation detection and root-cause analysis to identify the source of the violation.

3.4.3 A New Mechanism for End to End Latency Monitoring

We now present a mechanism for end-to-end latency monitoring that aggregates and analyzes metric streams for multiple components of the end-to-end latency.

Interface exposed to the control plane

The following abstractions are provided to the control plane of a platform service for using the end-to-end monitoring mechanism.

- **RegisterAppUnit.** Registration of an Application Unit when it is created. For instance, when a new topic is created in a publish-subscribe system, the topic should be registered as a new Application Unit. The Monitoring mechanism uses the identifier of the application unit to identify the metric streams that pertain to it, and perform end-to-end aggregation on them.

  Inputs:

  – AppUnitId: The ID of the application unit being registered.

- **RegisterMetricStream.** System components can register custom metric streams and publish their measurements. The metric streams are annotated with the application unit that they correspond do, as well as custom platform-specific tags.

Inputs:

– EntityID: The ID of the entity that this metric stream pertains to.

– MetricType: A string value denoting the type of metric stream, that is used for querying them.

– AppUnitId: The ID of the application unit that this metric stream belongs to.

– Labels: A dictionary containing a set of key-value pairs, each containing a platform-specific information about the metric stream.

– ValueSchema: A schema in YAML format that defines the data-structure of measurements for this metric stream. It is particularly useful for defining complex metric streams. Primitive datatypes can be represented by a string (e.g., int or str).

– BucketSize: The size of the bucket for data aggregation.

– AggregationFunction: The function to use for aggregation. The mechanism should support basic functions such as Mean, Median, Min, Max and percentile values.

Returns: A unique identifier for the metric stream, that can be used to record measurements and query measurements.

The following snippet shows an example, that measures the network latency of an instance of the pipeline stage shown in Fig. 3.16 to its downstream component. In the given platform service, the application unit that a metric stream belongs to is the downstream-most application component, or the root component. In this example, the application unit is the application component instance L_0.

```
-   entity_id : "L10"
    entity_type: "APP"
    app_unit: "L0"
    metric: "net_latency"
    labels:
        level: 0
    value_schema: float
    bucket_size: 5 secs
    aggr_fn: AVG
```

Each metric stream is represented as an ordered sequence of measurements along with their timestamps as shown in Eq. (3.3).

$$M = [\cdots, (t, v), \cdots] \text{ where } 0 < t < \infty \tag{3.3}$$

The mechanism time-aligns the measurements for a specific metric stream into well-defined buckets, that is necessary as a pre-processing step before multiple metric streams can be aggregated together. Since measurements from different metric streams are likely to be recorded at different instants in time, aligning them in time allows measurements from different metric streams to be processed together, such that two measurements from different metric streams that were collected at roughly the same time can be processed together[1]. This bucket-based time-alignment approach is an approximation of an ideal end-to-end aggregation of component latencies by recording measurements to every data-item generated by the data-plane. Such an ideal approach is infeasible for two reasons: (a) it behooves instrumenting the application components to record latency incurred by data plane actions which is neither desirable nor possible since the application logic is in the purview of the developer; and (b) it will vastly increase the amount of communication incurred by the system for monitoring the end-to-end latency. Therefore, we choose the aforementioned bucket-based approach. Fig. 3.15 illustrates the time alignment of a metric stream for a fixed size time bucket, and all measurements within a given bucket are aggregated by taking an average over them. The platform service is supposed to provide the bucket size B (in units of time) and the aggregation function $func$ for aggregating a metric stream M. As described in Eq. (3.4), the alignment of a metric stream involves aggregating all the measurements within a particular bucket interval using the provided aggregation function to form one measurement.

$$\begin{aligned} ALIGN_B^{func}(M) = [\cdots, (t, v), \cdots] &\text{ where } t = n \cdot B \text{ and } 0 < n < \infty \\ &\text{and } v = func(\{v' : (t', v') \in M \text{ and } B \cdot (n-1) < t' \leq n \cdot B\}) \end{aligned} \tag{3.4}$$

- **RecordMeasurement.** Platform service components can record measurements for a specific metric stream using the ID of the entity and the metric stream type.

[1]We assume that clock skew between the entities recording measurements is significantly less than the bucket size. This is a reasonable assumption because typical clock skews are not expected to be more than a few 10s of milliseconds

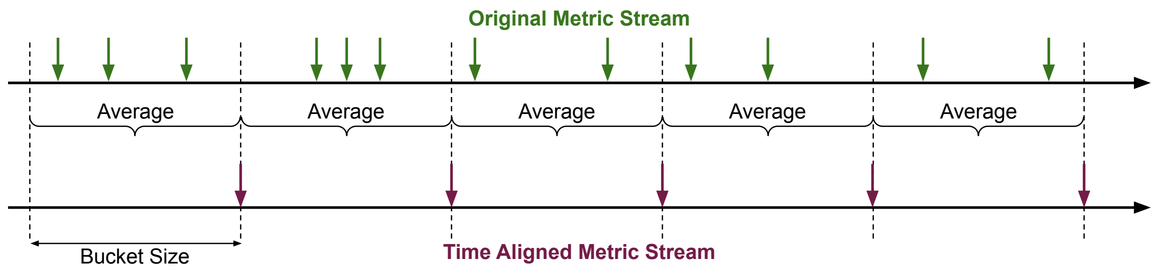


Figure 3.15: Illustration of time alignment of metric streams. The individual measurements (denoted by green arrows) within each time bucket are aggregated using the average function to generate the measurement for that bucket (shown by red arrows). The dotted vertical lines indicate the boundaries of buckets. The time alignment function takes as input a metric stream and time bucket size and returns another stream.

Inputs:

– MetricID: The ID of the metric stream that this measurement pertains to.

– Measurement: A YAML document containing measurement. It can be a nested YAML document in case the metric stream records complex values.

The parameters for an example call to the record measurement function are shown below.

```
-   metric_id: "net_latency_L0"
    value: 10.0
```

• **OnNewBucket.** This is a callback function that is called when time-aligned measurements of all the metric streams belonging to a particular application unit are ready for a given bucket. The receiver of this callback is supposed to be an entity in the control plane of the platform service, that receives the end-to-end aggregated view of all the metric streams associated with a particular application unit. The control plane entity uses these metric streams to make violation detection and reconfiguration decisions.

Inputs:

– AppUnitID: ID of the application unit that this set of metric streams belongs to.

– Bucket. The timestamp of the bucket that is being reported.

– Values. A collection of metric streams with their metadata and aggregated values for this bucket.

- **QueryMetrics.** The logic provided to the end-to-end monitoring mechanism via the ProcessMetrics interface should be able to query metric streams using their labels.

  Inputs:

  – MetricID: ID of the metric stream being queried.

  – AppUnitID: ID of the application unit being queried.

  – BucketTimeRange: A range of time within which all buckets for the given metric stream or application unit need to be returned.

  Returns: A list of buckets, each containing the timestamp of the respective bucket and a set of values for all the queried metric streams for that bucket.

Demonstration of using end-to-end monitoring for control plane policy of a platform service

In this section, we show how the end-to-end monitoring mechanism will be used for implementing a policy for detecting violations of end-to-end processing latency for application instances running on a geo-distributed Edge infrastructure, as part of the control plane of an application orchestrator. For this example, we restrict the discussion to clients and application instances for a single application unit, but the approach generalizes to multiple application units.

1. The library of the platform service running alongside each client and backend application instance generates two metric streams - namely the processing latency and network latency to the immediately downstream application component instance. For an application component e we denote them as $proc_e$ and net_e.

```
-   entity_id : "L20"
    entity_type: "CLIENT"
    app_unit: "L0"
    metric: "proc_latency"
    bucket_size: 5 secs
    aggr_fn: AVG
```

```
-   entity_id : "L10"
    entity_type: "APP"
    app_unit: "L0"
    metric: "net_latency"
    bucket_size: 5 secs
    aggr_fn: AVG
```

   The above listings show definitions of metric streams corresponding to the processing and network latency of the application component instance L_{20} and L_{10} respectively in the application instance shown in Fig. 3.16. Both these component latencies

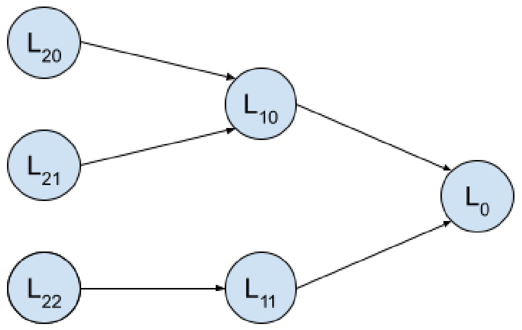


Figure 3.16: Schematic of a typical situation-awareness application. The application model resembles a tree, with the leaf vertices corresponding to clients. Each vertex that is not a leaf is an application instance running on the backend infrastructure (Edge-Cloud continuum). Each vertex processes data generated by the upstream vertices and sends its output data to the downstream vertex (if any).

pertain to the application unit that is associated with the "root" application component instance L_0. The platform service requests for time-alignment of all the metric streams using the function $ALIGN^{AVG}_{5secs}$.

2. The control plane policy for detecting violations of end-to-end processing latency receives the callback **OnNewBucket** for the application unit L_0 as soon as a new bucket for all metric streams belonging to it are available.

```
SELECT METRICS WITH
                entity_type="CLIENT" AND app_unit="L0"
```

3. The above set of metric streams are then grouped by the field $entity\_ID$ because they contain measurements for both processing latency and network latency for each client. Grouping them by $entity\_ID$ allows the policy to separate the metric streams of a specific entity from other entities.

4. For each client component c, the policy computes the set of application components S_c that process data generated by c. The following pseudocode illustrates this computation.

```
$n \leftarrow c$
$S_c \leftarrow \{\}$
while $n \neq \phi$ do
    $S_c \leftarrow S_c \cup \{n\}$
    $n \leftarrow M[n]$
```

We use the notation $M\left[n\right]$ to denote the downstream application component reading the output of n. The downstream application component for the root component L_0 is designated to be null (ϕ). This information is obtained from the application orchestrator platform service's control plane metadata about application mapping.

5. For each client c, now it is possible to compute a time-aligned metric stream that records the end-to-end processing latency for c.

$$E2E^*_c = \sum_{e \in S_c} proc^*_e + net^*_e \tag{3.5}$$

For those clients whose end-to-end latency estimation exceeds the application's threshold, a reconfiguration action is triggered. The objective of the reconfiguration action is determined by root-cause analysis, that is performed by the platform service's control plane policy itself (described in more detail in Section 6.5.4).

Briefly, the root-cause analysis analyzes the historical trend of each component latency that makes up the end-to-end latency (i.e., processing and communication latency of each component). It determines the relative contribution of each of the component latencies toward the end-to-end latency violation. In case the violation is most significantly caused by an increase in communication latency between an upstream-downstream application component pair $(l1, l2)$ then $l1$ is re-mapped to another downstream component instance that offers better network latency. Similarly, if the root-cause is an increase in processing latency at a certain component instance l, then its resource allocation is increased to bring down the processing latency. If resource allocation cannot be increased due to capacity constraints, a fraction of its immediately upstream component instances are mapped to another instance so as to reduce the workload on l.

Improvement over Previous Approaches

We evaluate the improvements brought about by incorporating the end-to-end latency monitoring mechanism into the monitoring policy. We perform the same experiment as in Section 3.4.2, with the difference being that the monitoring policy now uses the end-to-end latency estimate from the monitoring mechanism to detect violations. Fig. 3.17 shows the variation of number of migrations and number of violations experienced by each client during the experiment. Both these metrics are identical because the end-to-end monitoring

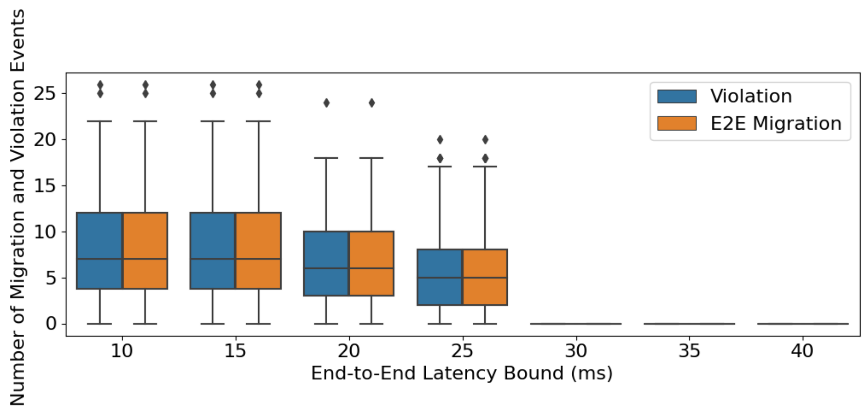


Figure 3.17: Number of migrations triggered by a monitoring approach that utilizes the end-to-end monitoring mechanism for detecting violations. Also shown are the actual number violations faced by clients. The end-to-end latency threshold is varied along the x-axis.

mechanism triggers a migration only when the end-to-end latency constraint is violated. Therefore no unnecessary migrations are triggered by the monitoring policy. A detailed design space exploration of the end-to-end monitoring mechanism will be discussed in Section 4.3.

Summary. This chapter presented three mechanisms that are needed by the control plane of platform services to efficiently operate on the Edge. The previous approaches of providing the functionality of these mechanisms are deficient, because of which we propose to design new mechanisms for the same. We propose the following new mechanism: (1) a dynamic spatial context management mechanism; (2) a network proximity estimation mechanism; and (3) an end-to-end latency monitoring mechanism. This chapter introduces the mechanisms, the interface they offer to the control plane policies and a summary of the improvements they can have on the functionality of control plane policies. We will discuss detailed design-space exploration for all three mechanisms in the next chapter.

CHAPTER 4
DESIGN SPACE EXPLORATION FOR IMPLEMENTATION OF PROPOSED MECHANISMS

In this chapter, we consider multiple design choices for implementing the mechanisms proposed in this dissertation. The aim of the design space exploration is to come up with the best design for each mechanism in the context of operation in a geo-distributed setting. The chosen design should be scalable with respect to the number of clients and Edge sites, should perform the desired functionality expected from the mechanism effectively and not result in high resource overhead on the scarce Edge resources.

4.1 Dynamic Spatial Context Management

We divide the design space exploration of the dynamic spatial context management mechanism into two parts. We first explore the appropriate choice of the spatial partitioning technique, that divides a geographical space into multiple tiles, and assigns spatial context to clients, application components and data-items. The second exploration we carry out is the system architecture for continuously monitoring client location and triggering a migration to a new tile when the client enters the new tile. We define the requirements expected from both these components of the architecture, enumerate the metrics of interest and carry out evaluations to quantify the performance of candidate design choices. Based on the results of the evaluations, we choose the design choice that performs best to build the dynamic spatial context management mechanism.

4.1.1 Client Workload assumed

The behavior of spatial context management is dependent on the locations of clients within the geographical space in question. We assume that an application would define a large geographical area (typically the size of a city) wherein its clients would be located, and the spatial context of a client would be a subset of the entire geographical area. For this design space exploration, we consider the area of downtown Shanghai where clients are spawned at random locations following a uniform probability distribution. We generate trajectories of 1000 clients that move in the geographical space following the random waypoint mobility model.

4.1.2 Maintaining Spatial Partitioning

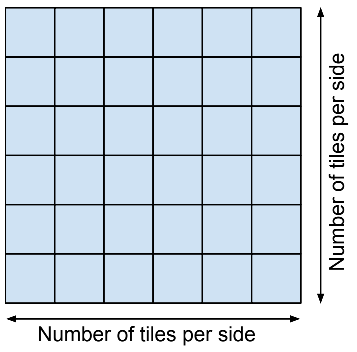
The objective of a spatial partitioning technique in the context of the dynamic spatial context management mechanism is to be able to associate multiple clients that belong to the same spatial context together. Each of the regions (that we call *tiles*) created out of spatial partitioning could be mapped to an application instance, in which case each application instance would serve the clients that belong to that tile. In another scenario, the spatial partitioning could serve as a range query lookup tool for spatially distributed data, and a tile could form the unit of data-sharing. We denote the number of entities (clients or data-items) mapped to a tile as the *occupancy* of that tile. In both these scenarios, a tile that has a very high occupancy would result in compute overload at the associated application instance or high latency overhead for executing range queries on the mapped data-items. Hence, the requirement is that the occupancy of each tile should be low enough so as to not cause performance degradation due to overload.

Metrics of Interest

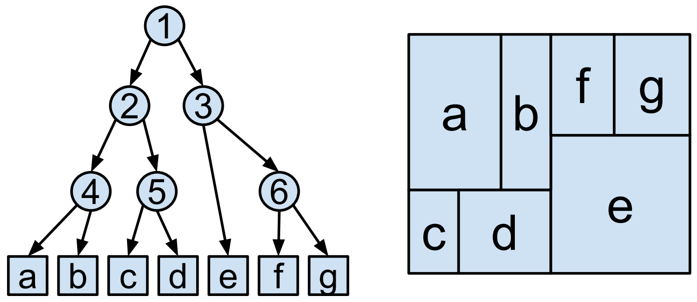
Considering a static scenario, with no mobility of clients, as an example, a trivial mapping that uses a very fine-grained spatial partitioning to map each entity to a distinct tile would result in the occupancy of each tile being minimal, because each tile would be associated with either exactly 0 or exactly 1 entity. However, this would also result in the existence of a large number of tiles, which is unnecessary and even counter-productive, because, for instance, applications do require that multiple clients be mapped to the same tile so that inter-client coordination can be possible. Furthermore, continuous client mobility would trigger very frequent requests for re-mapping clients to different tiles. Hence, we identify two metrics for quantifying the *goodness* of a spatial partitioning scheme.

- **Maximum Tile Occupancy.** We measure the maximum occupancy at any tile at any point of time, that would quantify the workload experienced by the associated application component or data-storage node.
- **Number of Active Tiles**. We measure the number of tiles being used at any given point of time, i.e., those tiles to which at least one entity has been mapped. Ideally, this number should be as low as possible, so that we do not have a large number of application instances or data nodes, that consume resources on the Edge.

For this evaluation, we are interested in evaluating spatial partitioning techniques using the aforementioned metrics, while assuming that the spatial context management mecha-



(a) Illustration of partitioning geographical space using the static partitioning technique.



(b) Illustration of partitioning geographical space using the KD-Tree technique.

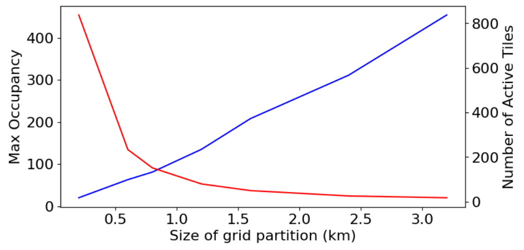
Figure 4.1: Illustration of the candidate spatial partitioning approaches evaluated in the design space exploration. In both the figures, the rectangular area represents the application's geographical coverage and each rectangle inside it represents a tile to which entities are mapped.

nism has accurate and real-time information about the location of each client. Following the Random Waypoint Mobility model as described earlier, the location of each client is updated every millisecond. The updated locations are fed to the spatial partitioning unit, that updates the client-to-tile mapping. At every time instant we calculate the two metrics of interest, hence generating a time-series for each of the spatial partitioning configurations evaluated. We later average these measurements across time and present a scalar value that represents the performance of the given spatial partitioning technique.

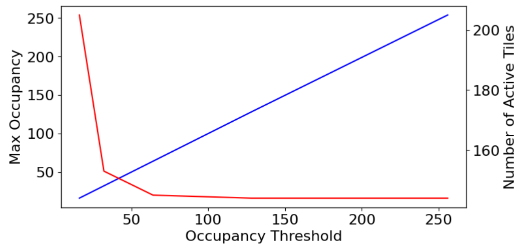
Candidate Design Choices Evaluated

We evaluate two spatial partitioning techniques that are common in the literature [33, 6].

- **Static Partitioning**. Similar to GeoHash, this geo-indexing technique statically divides the geographical space into a number of tiles. Since we are not interested in the numeric value of the tile's identifier (unlike typical use-cases of geo-indexing approaches such as GeoHash), we simplify the partitioning by assuming that geographical space is partitioned into a grid of squares, and the side length of each square is configurable. Each tile can then be represented by a tuple (row, col), where row and col represent the position of the tile in the grid. Given the location of a client, it is straightforward to map it to a tile based on the size of each tile and the size of the total geographical area.



(a) Variation of metrics of interest for the static partitioning technique with varying size of each partition.



(b) Variation of metrics of interest for the KD-Tree based dynamic spatial partitioning technique with changing occupancy threshold per tile.

- **KD-Tree based Partitioning** uses a two-dimensional KD-tree to partition geographical space. The vertices in the KD-tree represent geographical areas, with the root vertex representing the entire geographical space in question, while the leaf vertices represent the tiles. The spatial bounds of a vertex are fixed and decided when creating the vertex. Looking up the tile that a given client belongs to requires performing a traversal starting from the root vertex down to the leaf vertex that represents the actual tile. At each step in this traversal, the bounds of the child nodes and the given client's location are used to decide which of the child nodes to move to next. The size of each tile is not fixed, rather it changes dynamically as the tree is updated. The only constraint that is enforced by the tree is that the number of clients mapped to a specific tile should not exceed an *occupancy threshold*. If the occupancy of a particular tile becomes higher than the threshold, the tile is split and two children tiles are created. Furthermore, if the total occupancy of two tiles that have the same parent tile is less than the occupancy threshold, the two child tiles are merged.

We first evaluate the performance of the Static Partitioning technique against the aforementioned client workload. We evaluate the two metrics of interest - namely Maximum Occupancy and Number of Active Tiles. The evaluation is done with several different configurations of each partitioning technique. Fig. 4.2a shows the variation of the maximum occupancy and the number of active tiles as a function of the side-length of each geographical partition. As the size of the grid partition increases, the maximum occupancy grows, along with a decrease in the number of active tiles. This tradeoff is shown in Fig. 4.3 as well.

Next we perform the same experiment with the KD-Tree based spatial partitioning technique, wherein we configure the Occupancy Threshold for each tile. We vary the occupancy threshold from 8 to 32 and plot the metrics of interest in Fig. 4.2b. The metrics show a sim-

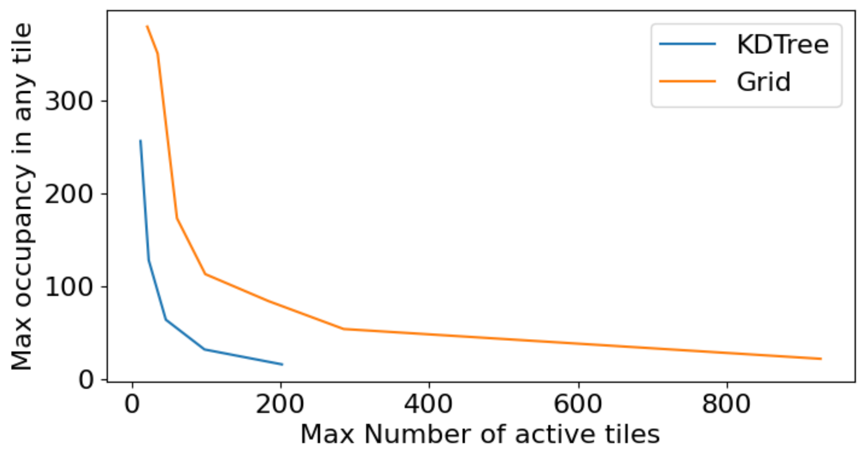


Figure 4.3: Tradeoff between the two metrics of interest - Maximum Occupancy and Number of Active Tiles - for the two types of partitioning techniques evaluated. We changed the configuration of each partitioning technique and repeated the experiment to obtain a range of performance output, that shows the tradeoff between the two metrics.

ilar behavior as in Fig. 4.2a. As the occupancy threshold increases, the number of active tiles that exist in the system decreases, because each tile can hold more clients. In addition, the increase of occupancy threshold also results in higher maximum occupancy in the system.

Fig. 4.3 shows the tradeoff between the two metrics of interest for the two spatial partitioning techniques that we evaluate. Both the partitioning techniques show similar behavior in the tradeoff curve, where an increase in maximum occupancy results in a decrease in the number of active tiles, and vice versa. However, the absolute values of both the metrics are much lower for the KD-Tree based partitioning than the Static partitioning. The KD-Tree partitioning outperforms Static partitioning because it builds tiles based on the physical distribution of clients, instead of being predefined as in the case of Static partitioning. Because of the adaptive nature of the KD-Tree partitioning, a smaller number of tiles are able to uniformly divide the clients.

The dynamically splitting and merging spatial partitions in the KD-Tree based spatial partitioning does not take place in the critical path of control plane operations. It is triggered asynchronously by the control plane policy when an opportunity to split or merge a tile is identified. When splitting and merging, the cache on the client side is invalidated and they connect to a new application instance that is created at the time of triggering the split or merge. The clients that connect to the new application instance would experience some downtime, because the new instance would take some time to initialize. However, such an

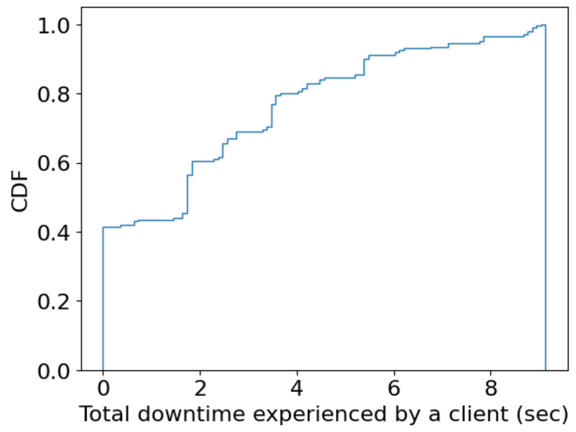


Figure 4.4: CDF of total downtime experienced by clients due to dynamic splitting and merging of spatial partitions by the KD-Tree based approach.

event does not occur frequently because split/merge operations are called in the event of spatial skews, which are infrequent. Fig. 4.4 shows the distribution of client downtimes experienced in one of the scenarios from the experiment performed to evaluate the KD-Tree candidate design, where the occupancy threshold was set to 32. Hence, we choose KD-Tree based dynamic spatial partitioning technique for building the dynamic spatial context mechanism.

4.1.3 Monitoring Client Location

Monitoring the current location of clients is necessary for maintaining the KD-Tree based spatial partitioning because of two main reasons: (1) it provides the information necessary for maintaining the occupancy of entities in different tiles, and trigger the remapping of an entity to another tile when it leaves the current one; and (2) when partitioning a tile, updated entity locations provide hints about how to partition the tile so that a roughly equal number of entities are present in both the children tiles. The metadata associated with the spatial partitioning is maintained in a centralized location that serves as the authoritative copy. Sending location updates from all clients to the centralized location consumes network bandwidth as well as creates a large number of occupancy updates that do not necessarily result in a change in the current tile. Hence, we conduct a design space exploration of the location monitoring module of this mechanism to find a design that lowers these overheads.

Metrics of Interest

We choose two metrics of interest for evaluating the candidate designs for the location monitoring module of the dynamic spatial context management mechanism.

- **Messaging Overhead.** We measure the number of location monitoring messages that need to be sent to the centralized location that maintains the authoritative copy of the spatial partitioning metadata. This metric should be minimized to improve system scalability.

- **Total Occupancy Violation Time.** We measure the sum of all the time durations for which some tile's occupancy threshold is violated. Since the occupancy threshold is set by the application, violating it would result in a workload surge leading to performance degradation.

Candidate Approaches Evaluated

- **Centralized Approach.** A straightforward design for maintaining updated client locations is to send all location updates from clients to the centralized location holding the authoritative copy of the spatial partitioning. Clients receive the identifier of the current tile based on their current location, and if the received tile identifier is different from the current tile, they would connect to the application instance corresponding to the new tile. The centralized spatial partitioning unit would transparently handle scenarios when the partitioning is updated due to tile merging or splitting operations. Although this approach is very straightforward to design and implement, it suffers from high overhead of constantly monitoring the locations of all clients at all times.

- **Distributed Approach.** Unlike the centralized approach, where the client simply reports the current location to the spatial partitioning module and receives the identifier of the current tile, the distributed approach maintains a cache of the spatial partitioning locally in the client library. The local cache is invalidated and updated every time the currently cached tile in the centralized authoritative copy is updated due to tile merging or splitting operations. The cache is then used by the client to keep track of its location with respect to the current tile, and when its location leaves the current tile, it triggers a migration to connect to the new tile. The cache also periodically reports its location to the authoritative copy so that tile split operations can

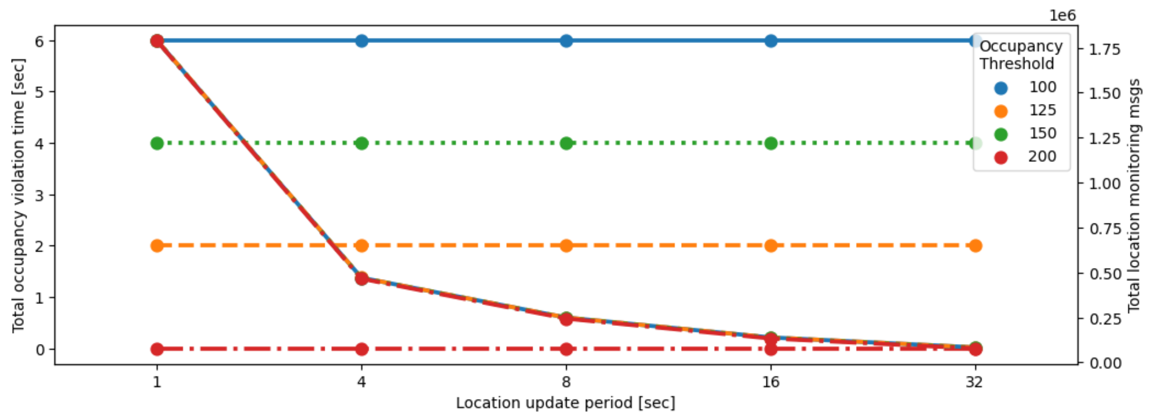


Figure 4.5: Effect of location update period on the duration of occupancy violation faced by tiles and the amount of location monitoring messages that need to be sent to the control plane.

effectively partition clients into the child tiles equally. The periodicity of reporting location to the control-plane is configurable, and it affects how uniformly clients are split among child tiles. We expect that with infrequent location updates from clients, the number of clients in child tiles as a result of a split operations would not be uniform, thereby increasing the likelihood of another occupancy threshold violation in the future.

Fig. 4.5 shows the variation of duration of occupancy violation and total number of location monitoring messages sent with increasing location update period. The increase in location update period does not have any noticeable impact on the violation duration, however it results in a significant drop in the number of messages for monitoring location of clients. Since the client immediately sends a location update when it leaves the current tile irrespective of the location update period, the violation duration is not dependent on location update period. In fact, for an occupancy threshold of 200, the violation duration is 0, that shows that there are no occupancy violations. Hence, choosing a high location update period does not significantly impact the total violation duration, but significantly reduces location monitoring traffic.

4.2 Network Proximity Estimation

Control-plane policies of platform services need to make data and compute placement decisions so as to satisfy data processing latency constraints. In an Edge setting, network communication latency forms a significant portion of the end-to-end latency. Previous works in the cloud computing domain have come up with accurate techniques for estimating computation latency, but did not consider communication latency due to the rather homogeneous and well connected nature of datacenter network topologies. In an Edge infrastructure, it is important to be able to accurately estimate the network latency between a pair of system entities, so that the control plane policy can take a data/compute placement decision to satisfy end-to-end latency.

Network proximity estimation cannot be done using active measurements at the time of execution of the control plane policy logic, because a typical policy evaluates several candidate nodes (such as selecting the best candidate for application placement) and therefore requires pairwise network latency between several pairs of nodes to take its decision. Waiting for the measurements to complete in the critical path of policy execution would severely impact the responsiveness of the control plane. Hence, the network proximity estimation mechanism needs to continuously maintain network proximity metadata for each node in the infrastructure. The size of the metadata needs to be tractable so that it can be efficiently queried to estimate pairwise network latency.

The mechanism should estimate pair-wise network latency with high accuracy. Furthermore, it should be able to quickly adapt to changes in network topology due to client mobility or link failures. Both the accuracy and recovery-time of the mechanism to topology dynamism should not deteriorate with increasing number of nodes, meaning that the mechanism should be scalable.

4.2.1 Metrics of Interest

We consider the following metrics of interest when evaluating the candidate design choices for network proximity estimation mechanism.

- **Latency Estimation Error.** We measure the root-mean squared error between the estimated and the actual (ground-truth) network latency. For a pair of nodes i and j, the estimated and actual network latencies for the pair (i, j) is denoted by $N_{(i,j)}$ and $\hat{N}_{(i,j)}$, respectively. The root-mean squared error (RMSE) in latency estimation is then calculated as shown in Eq. (4.1).

$$RMSE = \sqrt{\sum_{\forall (i,j) i \neq j} \left(N_{(i,j)} - \hat{N}_{(i,j)} \right)^2} \tag{4.1}$$

- **Number of Messages Exchanged**. We measure the number of messages exchanged

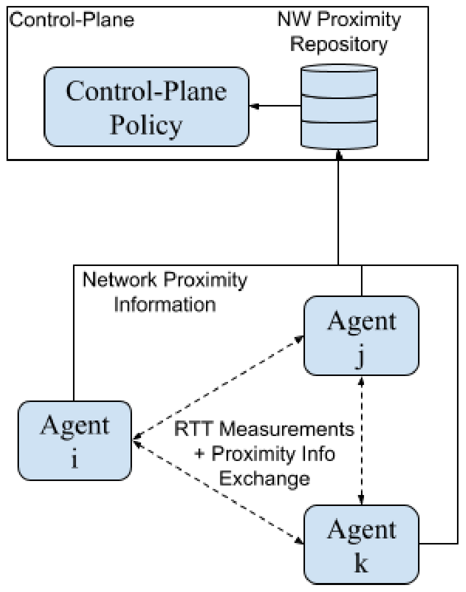


Figure 4.6: Basic components in the system architecture of the network proximity estimation mechanism. Multiple participating agents communicate among each other to perform actual RTT measurements and compute network proximity, and share the proximity information with one another. The network proximity information is uploaded to a repository in the control plane from where this information is supplied to control plane policies.

between the various nodes in the system before the error in latency estimation drops below the maximum threshold.

- **Amount of Metadata** needed by control-plane policies. We analytically calculate the amount of information that would need to be stored at the control plane in order to support a typical compute/data placement policy.

4.2.2 Candidate Design Choices Evaluated

Fig. 4.6 shows the basic architecture that the various design choices of the network proximity estimation mechanism have. It is composed of a number of *agents*, with one agent co-located with every system entity with which network proximity needs to be measured. Each agent computes its network proximity to the other agents, and communicates the proximity information to a central repository, that is queried by control-plane policies for estimating network round-trip time (RTT) between any two system entities. The design choices that we explore in this section follow the agent-based architecture, and differ in the kind of communication protocol that each agent follows and the kind of metadata stored by the mechanism that allows it to answer network proximity queries. We evaluate the following design choices for the network proximity estimation mechanism.

Pair-wise Measurements based Approach

In this approach, representation of network proximity is in the form of a 2-dimensional array A, where $A[i, j]$ represents the network latency between agent i and j. Each participating agent periodically performs a network round-trip time (RTT) measurement between itself and another agent. Each agent uses a round-robin policy to select which other agent it is going to probe to measure the RTT. After each measurement, the agent provides the RTT information between itself and the other agent with which the measurement was performed to the centralized repository of network proximity.

Network Coordinates

Network coordinate (NC) systems are distributed protocols to scalably determine the network proximity between a pair of nodes in a distributed system without performing direct measurements [34] between all pairs of nodes. Such systems embed nodes in a geometric space such that the network latency between any two nodes can be estimated by calculating the Euclidean distance between their positions (coordinates) in this space. Previous work on analysis of latencies in the Internet has shown that nodes can be embedded in a 3-dimensional or higher space with relatively high accuracy of network proximity estimation [35].

Embedding in d-dimensional space. Each agent i maintains a network coordinate x_i that is an d-dimensional vector. Each agent i periodically performs an RTT measurement with another agent j and also fetches the current network coordinate of agent j, that we denote as x_j. By using the measured actual RTT $rtt_{i,j}$ between itself and the other agent j, the given agent i updates its own network coordinate so as to reduce the error of latency estimation. To do so, agent i first calculates the error in latency estimation, as shown in Eq. (4.2).

$$e = rtt_{i,j} - ||x_i - x_j|| \tag{4.2}$$

Each iteration of this coordinate update process at agent i aims at applying a force on x_i so as to move it toward its correct position in the d-dimensional space with respect to agent j's coordinate. In other words, if the error calculated in Eq. (4.2) is positive, x_i would be pushed away from x_j, otherwise it will be pulled closer to x_j. This notion is captured in Eq. (4.3), that computes the unit vector of the force to be applied to x_i.

$$dir = u\,(x_i - x_j) \tag{4.3}$$

The force that needs to be applied to coordinate x_i is in the direction of the unit vector in Eq. (4.3) and has the magnitude proportional to the error in Eq. (4.2). x_i is then updated in the direction of the force by a small and configurable amount.

$$x_i = x_i + \delta \cdot e \cdot dir \tag{4.4}$$

Capturing Access Link Delays. A number of hosts connected to the Internet today do so behind access links. While the d-dimensional Euclidean space is good at modeling latencies in the Internet core, incorporating a scalar $height$ component in the network coordinate significantly improves the network RTT estimation error [36]. The height component represents the network latency incurred to traverse the access link beyond which latencies can be estimated using the Euclidean coordinate system. The estimated RTT between two agents i and j with combined network coordinates $(x_i, height_i)$ and $(x_j, height_j)$ respectively is given by $||x_i - x_j|| + height_i + height_j$. This equation captures the fact that for packets to travel from agent i to j, they first have to traverse the access link of agent i, travel through the Internet core toward agent j (that can be modeled using Euclidean distance), and then traverse the access link of agent j.

Reducing Errors due to Triangle Inequality Violations. Internet topologies frequently violate the triangle inequality that should ideally hold in a Euclidean space. The triangle inequality requires that the sum of the RTT between agents i and j and that between agents j and k should be greater than the RTT between agents i and k. This inequality is violated in real-world network topologies because of heterogeneous routing policies [37]. However, Lee et al. [35] found that such violations occur more frequently among nodes that are at closer network distances from one another. Hence, they introduce a scalar $adjustment$ term in the network coordinate to account for the non-Euclidean effect due to triangle-inequality violations. The adjustment term is calculated as shown in Eq. (4.5), where n represents the number of measurements taken.

$$adj_i = \frac{1}{2} \cdot \frac{\sum_j rtt_{i,j} - ||x_i - x_j||}{n} \tag{4.5}$$

We employ a popular decentralized network coordinate protocol, Vivaldi [36] with some enhancements proposed by Ledlie, et al. [38] and Lee, et al. [35]. Prior art has shown that network coordinate protocols provide efficient, accurate, and stable latency estimates in the wild [38].

4.2.3 Evaluations of Candidate Design Choices

We evaluate the performance of the candidate design choices for implementing the network proximity estimation mechanism and present the results in this section.

Accuracy of Network RTT Estimation

The network proximity estimation mechanism is expected to accurately estimate network RTT between agents in a real-world geo-distributed infrastructure topology. We use the topology of Edge sites belonging to Alibaba Edge Node Service [13], that has sites deployed all across mainland China. The dataset provides city-level location of Edge sites along with the actual network RTT between them. In this experiment, in addition to evaluating the error of RTT estimation, we also intend to study how the scale of the infrastructure topology affects the error. We build infrastructure topologies of increasing scale by selecting the top-k cities with the most edge sites and only considering the sites in those cities. Each site runs an agent of the network proximity mechanism, and communicates with potentially every other Edge site to collect RTT and update its network proximity model.

Fig. 4.7 shows the evolution of the error in RTT estimation over time for network topologies of increasing scale. The root-mean squared error of latency estimation does increase with an increase in the scale of the network topology, but it converges at around 4.5ms in RTT estimation, which is a reasonable error rate given the variance in ground-truth latency measurements. We do not evaluate accuracy of the pairwise measurements based approach, because the RTT estimates are simply an aggregation of the previously measured RTT values, and hence it would always result in very low error.

Communication Overhead

We now evaluate the amount of communication that needs to happen between the agents of the network proximity estimation mechanism in order to build a reasonably accurate inter-agent network RTT estimation model. We compare the number of messages that need to be communicated between the agents for both the pairwise measurement and network coordinates based designs of the mechanism.

For this experiment, we intend to evaluate network topologies of much larger scale than the Alibaba Edge Node Service topology used in Section 4.2.3. We adopt a hub-and-spoke model for generating a synthetic topology for this experiment, with each spoke having a network latency uniformly sampled between 10 and 60 ms. We choose such a

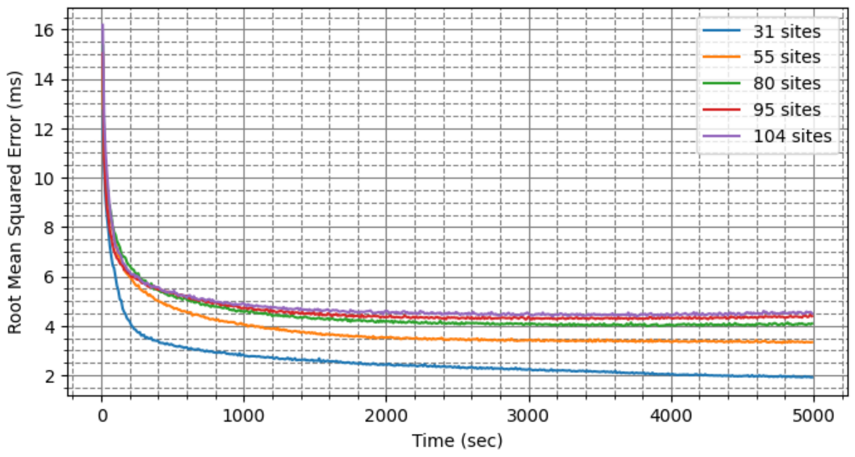


Figure 4.7: Evolution of error in RTT estimation for the network coordinate-based design. The figure shows the RTT estimation error for network topologies of increasing scale.

model for building the topology because it offers a simple way to model random network characteristics between different pairs of nodes. Each node in the topology runs an agent of the network proximity estimation mechanism. The number of nodes in the topology is varied to evaluate the design choices at different scales.

Fig. 4.8 shows the number of communication rounds that need to take place (combined over all nodes) before the network proximity estimation mechanism's error falls below a threshold of 8 milliseconds. The pairwise measurement approach needs to measure the network latency between all pairs of nodes, meaning that it requires $N\,(N-1)$ rounds of communications, that grows quadratically with increasing number of nodes. On the other hand, the network coordinates approach grows almost linearly, because it only tries to embed nodes in a high-dimensional space based on a small set of measurements. Hence, the network coordinates based approach is a more scalable design for network proximity estimation.

Size of Network Proximity Repository

Control-plane policies would frequently query the centralized repository of network proximity estimation to obtain the RTT between a pair of agents. A large metadata would mean that the query would take longer to execute, hence, reducing the speed of control-plane policy execution. We compare the amount of metadata to be stored for the two design choices.

Fig. 4.9 shows how the metadata size grows with increasing number of agents. Since the pairwise measurements based design would need to store the network latency between

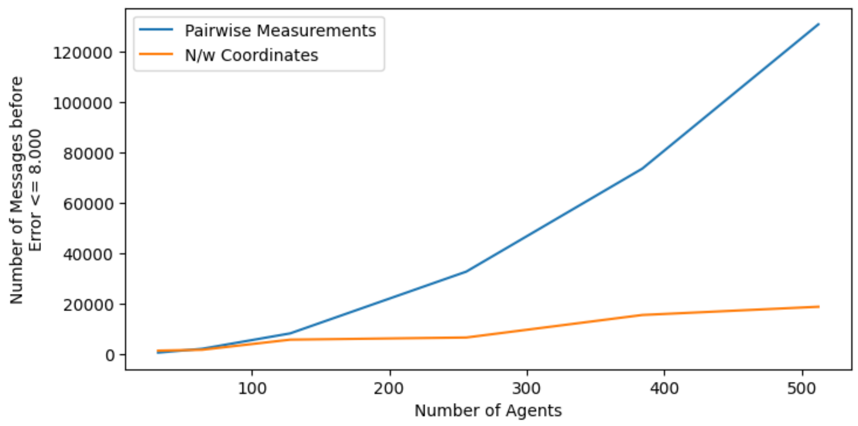


Figure 4.8: Communication overhead of the two design choices for network proximity estimation mechanism. The pairwise measurement based design requires $O\left(n^2\right)$ communication rounds, whereas the communication rounds needed by the network coordinates based approach scales almost linearly.

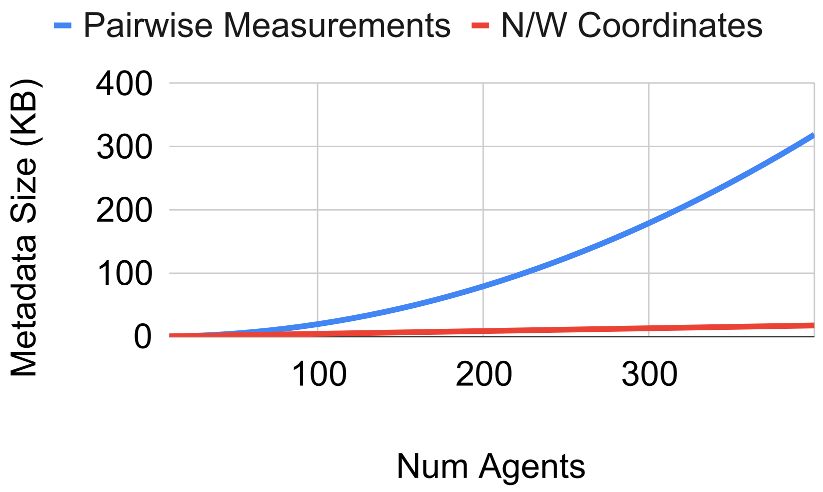


Figure 4.9: Comparison of the size of network-proximity metadata that needs to be stored at the control-plane.

each pair of agents, the size of the metadata grows quadratically. However, the metadata for the network coordinates based approach consists of one coordinate per agent, that amounts to 44 bytes and grows linearly.

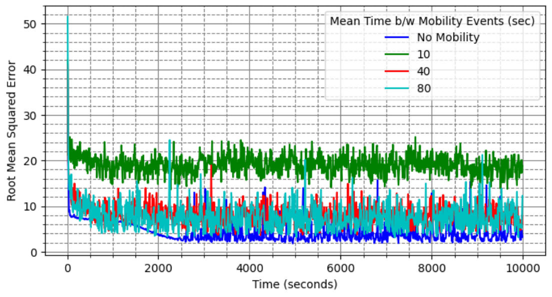
As we show in Chapter 5 and Chapter 6, each client device and Edge site run a network proximity agent. Therefore under typical Edge computing scenarios, we expect the number of agents deployed atop Edge sites and clients to range in the thousands. At such a scale, the network coordinates based approach significantly outperforms the pairwise measurements based approach in terms of number of messages exchanged and amount of network proximity metadata to be stored in the entity that manages the control policy.

4.2.4 Running Network Coordinate Agents on Mobile Clients

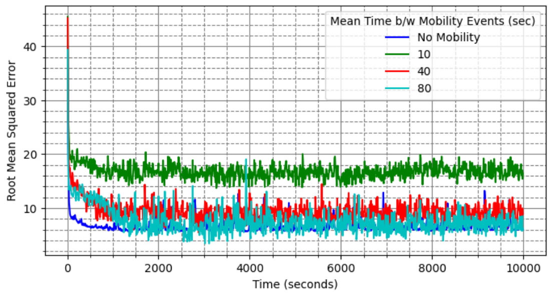
Client mobility results in the change of network routing between client and the Edge infrastructure, since the network access point to which the client is connected changes. The change of network access points results in a change in network latencies to reach the Edge sites, that affects the ground-truth on which the network coordinates protocol relies for converging to stable coordinates. The end result of these perturbations is a deterioration of the RMSE of network latency estimation, that is shown in Fig. 4.10. To evaluate the performance of the network coordinates protocol used in the Network Proximity Estimation mechanism, we evaluate the protocol with varying degrees of client mobility. Client mobility affects network coordinates protocol primarily by continuously changing the network access point used by clients to connect to the Edge infrastructure. Hence, we emulate different speeds of client mobility by controlling the average time between network access point changes for each client. The time between two consecutive access point changes for a client is sampled from an exponential distribution with the mean of the distribution set as mentioned above. For all scenarios with different number of clients, the RMSE in network latency estimation is the lowest for the case without mobility and increases progressively with more frequent mobility events per client. Hence, we conclude that the frequent changes in network connectivity for clients results in deterioration of RMSE of network latency estimation. Therefore, we propose an approach to eliminate this error in the following subsection.

Network Coordinate Proxy running on Edge Gateway

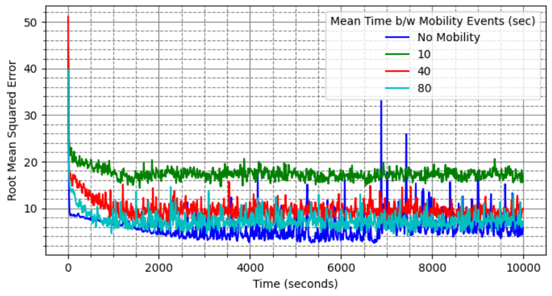
Mobile devices invariably connect to the Internet via a nearby gateway node that runs on the Edge of the network e.g., local breakout [39] for clients running on a 4G/LTE network.



(a) 64 Clients.



(b) 256 Clients.



(c) 512 Clients.

Figure 4.10: Variation of RMSE over time for varying number of clients and varying degrees of client mobility.

We assume the presence of a lightweight network coordinate proxy (NC Proxy) running on such a gateway node, serving as the source of network coordinate information for the mobile clients connected to that gateway node. For a client that is connected to the Internet via a particular Edge gateway, all uplink and downlink traffic flows through that gateway, and it is located at a fixed access network latency from the client. Hence, the network latency between the client and an Edge site can be calculated by computing the sum of the network latency between the site and the Edge gateway and the access latency between the client and the gateway. Therefore, the network coordinate agent on the mobile client computes its current coordinate by using the vector and adjustment fields of the gateway's coordinate, and adding the access latency to the height component of the gateway's coordinate. The access latency is monitored by periodic measurement from the gateway. It is noteworthy that the number of such network coordinate proxies is going to be much smaller than the number of clients – because each proxy serves a large number of clients. Hence aforementioned strategy for calculating network proximity is not only more stable for mobile clients, but also more scalable.

In this dissertation, we assume that the accurate discovery of the network coordinate agent running on the current serving Edge gateway can be facilitated by a DNS-based

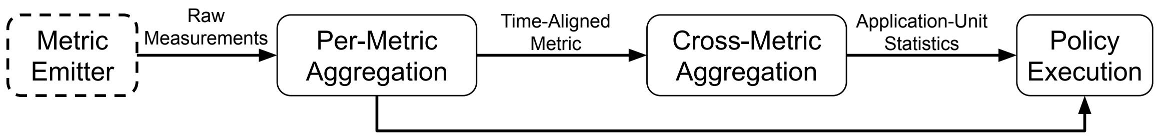


Figure 4.11: Logical components in the end-to-end monitoring mechanism and their interactions.

mechanism. Such a mechanism for application-level resolution at the Edge has been already presented in the context of content-delivery networks [40].

4.3 Distributed End-to-End Monitoring

The data-plane of applications using a platform service usually comprises multiple components, with each component performing a specific action, and incurring latency. The latency incurred by each such component adds toward the end-to-end latency of the application, that is expected to be lower than a certain threshold by the developer. Hence, the end-to-end monitoring mechanism aims to provide the ability to monitor the end-to-end latency of the data-plane for a variety of applications running on different platform services. We propose to do so by measuring all individual component latencies independently. The collected measurements of these metrics streams are then aligned with respect to time using their timestamps and summed up to compute the end-to-end latency. The end-to-end latency estimate can then be used to check whether the constraints specified by developer have been violated. In the case of detecting a violation, the measurements of individual component latencies are used for performing a root-cause analysis to detect the component latency that is the source of the violation.

4.3.1 Logical Components in the Monitoring Mechanism

Fig. 4.11 shows the logical functions that we propose as a part of the end-to-end monitoring mechanism. These functions perform all the necessary actions that are needed by typical platform services in order to continuously serve applications' desired quality of service.

Metric Stream Emitter

The Metric Stream Emitter component represents the platform service component generating measurements for a metric. The Metric Stream Emitter could be the client library on a certain client device, an application instance or a platform service component running on an Edge site.

Per-Stream Aggregation

The Metric Stream Emitter generates raw measurements that are processed by the Per-Stream Aggregation function, that time-aligns the metric measurements while performing an aggregation as well to reduce the data volume. The output of the per-metric aggregation component is a stream of time-aligned measurement values that are emitted at regular intervals. The time interval between two consecutive measurements emitted by the per-metric aggregation component is equal to the bucket-size parameter of the time-alignment process. The timestamp associated with each aggregated time-aligned measurement values in the output stream is the start timestamp of the bucket.

Cross-Stream Aggregation

The Cross-Stream Aggregation function takes multiple time-aligned metric streams at a particular timestamp as input and runs a platform-specific function to generate an aggregated value for that timestamp.

Policy Execution

The control-plane policies then read the aggregated statistics generated by cross-stream aggregation as well as the time-aligned metric streams to make decisions such as whether a certain application instance is undergoing performance violation and subsequently determining the root-cause of the violation.

4.3.2 Design of the core components of end-to-end monitoring mechanism

Metrics Agent

The Metrics Agent component is the one that is deployed alongside application and platform service components that need to be monitored, and acts as the interface between the system components and the monitoring system. It is deployed as part of the client library or the application runtime or as a side-car of the platform service components. It provides interfaces for application and platform service components to register metric streams and record measurements for those metrics.

Metrics Server

The Metrics Server is the component that holds the metric streams reported by the various Metric Agents. The Metric Server serves two main functions - (i) it acts as a store for time-series measurement values, and (ii) an execution runtime for the aggregation and policy functions as discussed in Section 4.3.1.

The Metrics Server provides an interface similar to a Time-Series Data Base (TSDB). The Metrics Server provides a relational schema for defining metric streams and their associated metadata. Having a relational schema also allows aggregation and policy logic to express queries based on metadata fields. The Metrics Server supports high-velocity ingestion of measurements, and efficient support for querying historical metric values.

The Metrics Server supports platform-specific functions to be executed over the monitoring data that is stored in its data store. It allows the aggregation and policy functions to query metric streams using their metadata fields. Once the identifier for a particular metric stream is known, the Metrics Server allows queries to fetch most recent as well as historical measurements recorded for that metric stream.

4.3.3 Metrics of Interest

We consider the following metrics of interest for quantifying the *goodness* of the various design choices for implementing the end-to-end monitoring mechanism.

- **Monitoring Traffic through WAN.** The large number of application and platform-service entities continuously processing data and generating performance metrics would present a large volume of data to be fed into the monitoring system. The amount of traffic sent through the WAN needs to be minimized for the mechanism to be scalable.

- **Elapsed Time for Detecting Violation.** The time taken by the monitoring subsystem (including the Policy) to detect a significant change in the measurements of a metric stream defines the responsiveness of the platform service to alleviate a potential violation. Hence, we intend to minimize the time taken by the monitoring mechanism to detect a violation.

4.3.4 Design Choices

We explore two design choices for organizing the logical functions discussed in Section 4.3.1 over a geo-distributed infrastructure. We consider two design choices.

Fully Centralized Approach

In the fully centralized approach, all of the monitoring functionalities, ranging from the Per-Stream Aggregation to the Policy Execution are hosted in the Cloud. In this approach, the measurements of metric streams are emitted by Metric Emitters located at the Edge within Metrics Agents, and the raw measurements traverse through the WAN to get to the downstream functions.

Distributed Approach

In this approach, the Metric Emitters are located within Metrics Agents, that is same as the centralized approach. In addition to that, the Per-Stream Aggregation function for a given metric stream is co-located with the Emitter for that metric stream. In this way, raw measurements are not sent through the WAN, rather it is the time-aligned metric stream that is sent to the Cross-Stream Aggregation function. Since the bucket-size configuration parameter of a Per-Stream Aggregation instance does not change frequently, it is possible to maintain several geo-distributed instances at a large scale without the overhead of reconfiguring them frequently.

4.3.5 Evaluation of Candidate Design Choices

We evaluate the aforementioned candidate design choices for implementing the end-to-end monitoring mechanism and present results in this section. In our evaluations, we consider one application unit that has a number of distributed system entities associated with it. Each of these system entities generate measurements to one metric stream at a certain measurement frequency. Each metric stream is time-aligned using a specific bucket-size wherein the measurements falling inside a bucket are summarized by taking an average. The time-aligned summaries of each metric stream are sent to the Multi-Stream Aggregation that checks if the measurement summaries of all the metric streams for the current bucket have been received, and if so, passes them to the policy execution component.

In our experiments, the Metric Emitters are located on the Edge, while the Cross-Stream Aggregation is deployed in the Cloud. Per-Stream Aggregation can run either on the Edge alongside Metric Emitters or in the Cloud based on the design choice being evaluated.

Fig. 4.12 shows the network bandwidth usage by the two design choices we evaluate for the end-to-end monitoring mechanism. We perform the evaluation for two bucket sizes - 1 second and 4 seconds - and two per-stream measurement rates - 2.5 and 10 measurements

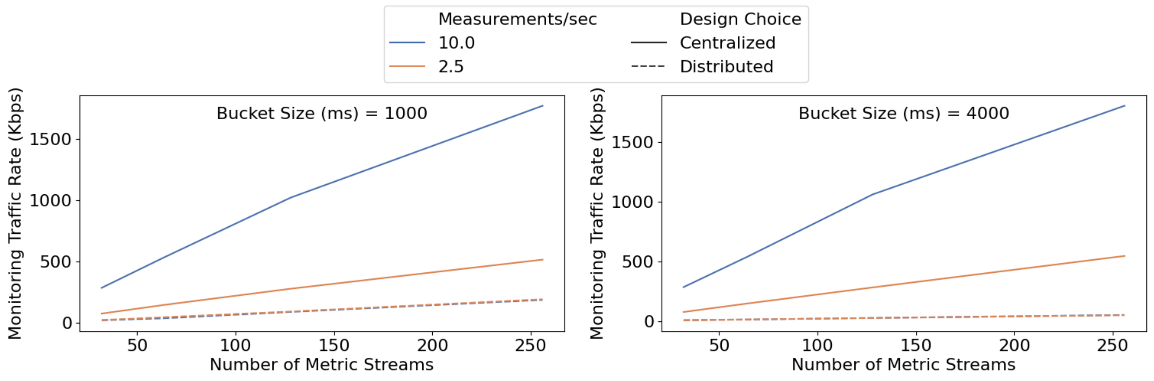
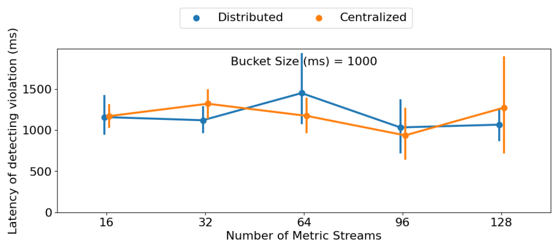


Figure 4.12: Network bandwidth usage of the design choices for end-to-end monitoring mechanism.

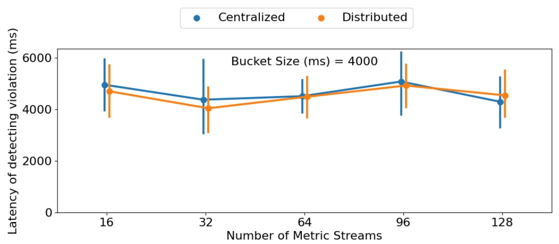
per second. For all four of these configurations, we see that the Distributed design choice incurs less network traffic between the Edge and the Cloud as compared to the Centralized design choice. Furthermore, the traffic consumed by Distributed approach does not change with different per-stream measurement rates because it only sends a summary of measurements (average) at an interval of 1 bucket size. This result quantitatively affirms our intuition that the Distributed design would be more scalable in terms of network bandwidth usage. We demonstrate the utility of the end-to-end monitoring mechanism in the context of a publish-subscribe system in Chapter 5. For the UAV swarm coordination application, each UAV creates a distinct metric stream. Similarly in Chapter 6, we present the use of the proposed monitoring mechanism in the context of an application orchestrator. In the case of the collaborative perception application, each client device as well as each instance of an application component generates one metric stream. Therefore, in typical application scenarios with thousands of clients in a city, we expect to see several thousand metric streams being processed by the monitoring mechanism for that city alone – leading to a significant difference in the network traffic usage between the two design choices.

Fig. 4.13 shows the variation of the elapsed time for detecting a violation with the two evaluated design choices, with the per-metric-stream measurement rate of 10 measurements per second. The results show that both Centralized and Distributed Approaches have a similar behavior in terms of the violation detection latency, with the latency being dependent only on the bucket size.

Hence, we conclude that the Distributed Design choice is not only more scalable in terms of network usage, but it also provides the same violation detection latency as the Centralized approach. Hence, we use the Distributed design choice for implementing the end-to-end monitoring mechanism.



(a) Elapsed time for detecting violations with a per-stream alignment bucket size of 1 second.



(b) Elapsed time for detecting violations with a per-stream alignment bucket size of 4 seconds.

Figure 4.13: Variation of the elapsed time for detecting a violation with increasing number of metric streams to be aggregated. The Centralized and Distributed approaches incur similar time to detect a violation, which remains relatively constant with increasing number of streams and only depends on the bucket size.

Summary

So far in this dissertation, we have determined the three mechanisms which are its main contributions. We have identified the right abstractions that they should offer to the control plane policies and performed a design-space exploration to identify the best design for each mechanism for an Edge infrastructure. The next step is to demonstrate the utility of the proposed mechanisms in constructing platform services that will be used by application developers in constructing platform services to serve situation-awareness applications. We do so through three platform services, that are enumerated as follows:

- **OneEdge** is an application orchestrator, specifically designed for situation-awareness application to operate on Edge infrastructure. Chapter 6 describes **OneEdge** and its use of the three mechanisms for placement of applications on the infrastructure, mapping clients to application instances and monitoring the performance of deployed application instances.

- **ePulsar** is a topic-based publish-subscribe system that provides guarantees on end-to-end message delivery latency. It is presented in Chapter 5. It leverages the network proximity estimation mechanism to place topics on brokers nodes to ensure latency constraint satisfaction. It uses the end-to-end monitoring mechanism to detect when a certain topic's latency threshold is violated.

- FogStore is a key-value store that provides a tradeoff between latency and consistency while ensuring tolerance from geographically correlated failures (which are more likely in an Edge infrastructure). It is presented in Chapter 7, wherein we show

how FogStore leverages the Dynamic Spatial Context Management mechanism for placement of data replicas among the data nodes and choosing the right consistency level for serving clients to provide the right tradeoff between latency and consistency.

CHAPTER 5
EPULSAR: TOPIC-BASED PUBLISH-SUBSCRIBE SYSTEM WITH END-TO-END LATENCY GUARANTEES

Applications such as navigation control of Unmanned Aerial Vehicles (UAVs) and collaborative perception for autonomous driving need to support a large number of clients, retaining high throughput and low latency communication. The synchronization decoupling provided by publish-subscribe (pub-sub) systems [41, 42] makes them an ideal messaging middleware for supporting these applications. Typically, pub-sub systems consist of "broker" *middleware nodes* that are responsible for message exchange between "producers" and "consumers" of data in the system. Brokers manage "topics", where a topic acts like a message queue to which producers publish messages, and consumers receive messages from it. Popular pub-sub systems such as Apache Kafka and Pulsar are commonly used for supporting low latency and high throughput messaging for cloud applications. Pub-sub systems have been shown to be suitable for sharing game state updates in MMOGs [42], swarm synchronization for autonomous robots (drones) [43], and data distribution for large-scale stream processing [44]. However, contemporary applications such as large scale IoT, and Unmanned Aerial Vehicle (UAV) coordination pose latency constraints that make cloud-based publish-subscribe system deployments unsuitable due to the high WAN latency between clients (producers and consumers) and middleware (broker) nodes. Given the proximal nature of Edge resources, they can be utilized for hosting pub-sub middleware nodes close to clients and thereby provide low end-to-end message delivery latency.

However, adapting state-of-the-art cloud-based pub-sub systems like Kafka and Pulsar to a geo-distributed Edge infrastructure poses peculiar challenges. Such systems typically are topic-based and they partition topics among brokers by computing a consistent-hash of the topic name. Consistent hashing ensures even distribution of load among brokers - that is key to manageable end-to-end latency in datacenters where the network topology is more or less homogeneous. However, in Edge infrastructure, the physical location and network connectivity of a client have a significant impact on the client-broker network latency, and latency-agnostic consistent hashing does not account for network proximity. In addition, client mobility requires constant adaptation of the broker hosting a given topic so as to continuously provide low end-to-end latency for that topic. The network infrastructure itself could experience changes (e.g., increased latency between servers) that might affect end-to-end latency. Finally, due to the capacity constraints and limited statistical multiplexing at Edge sites, load-aware topic partitioning is important to avoid workload hotspots and minimize end-to-end latency [25].

To address the above challenges, we design a novel *edge-centric control plane architecture* for pub-sub systems. The elements of this architecture include the following features:

1. **ePulsar** leverages the Network Proximity Estimation mechanism for its latency and load-aware broker selection policy that assigns topics to brokers to ensure that end-to-end message delivery constraint of each topic is satisfied.

2. **ePulsar** leverages the End-to-End Monitoring mechanism to monitor client and broker network proximity metrics as well as per-topic traffic characteristics and aggregates that information for each topic. The per-topic aggregates of monitoring data are then processed by control plane policies for detecting if a topic needs to be migrated away from the current hosting broker to alleviate latency violation.

The roadmap of this chapter is as follows. Section 5.1 presents the necessary background for this chapter. Section 5.2 presents the architecture of **ePulsar** . Section 5.3 describes how **ePulsar** uses the Network Proximity Estimation mechanism for estimating the end-to-end message delivery latency for broker selection. Section 5.4 presents **ePulsar** 's use of the End-to-End Monitoring mechanism for detecting and alleviating violations of end-to-end message delivery latency constraint. Section 5.5 presents implementation details in **ePulsar** and Section 5.6 presents results of experimental evaluation of **ePulsar** . Finally, Section 5.8 presents concluding remarks.

5.1 Basics of Publish-Subscribe

5.1.1 Publish-Subscribe Communication Model

The publish-subscribe communication pattern is used widely across the current software ecosystem [41], especially when there are going to be a large number of communicating entities. In the publish-subscribe pattern, there are two classes of communicating entities for a given type of data - *publishers* and *subscribers*. As their name indicates, publishers generate data that is consumed by subscribers. The main motivation for using a publish-subscribe system is that it does not require the publishers to know who the subscribers are so that the data can be sent to them. Instead, a typical publish-subscribe system allows the subscribers to express their interest in a certain type of data, and whenever a publisher generates a data-item that matches a subscriber's interest the subscriber is notified. This data exchange is done by using a set of middleware nodes that form the core of the publish-subscribe system, and are typically called *brokers*. A publisher sends each data-item that it generates to a broker, that then determines the set of subscribers who should be notified of this data-item, and sends the data-item to them. By relying on brokers, a publish-subscribe system decouples the data producers and consumers. This decoupling can be categorized into three classes:

- **Space decoupling**: The communicating entities do not need to know about each other. In other words, the publishers do not need to hold a reference of the subscribers and similarly, subscribers do not need to hold a reference of the publishers. They simply connect to the broker and send/receive messages from it.

- **Time decoupling**: For a specific message, the publisher generating it and the subscribers consuming it do not need to be active at the same time. Subscribers are notified of a messages that matches its interest even though it was published when the subscriber was not connected to the publish-subscribe system.

- **Synchronization decoupling**: Publishers are not blocked for producing messages by the subscribers consuming them. Subscribers can keep performing other tasks in parallel while they are asynchronously notified of incoming messages from publishers.

There are three broad categories of publish-subscribe systems [45]:

- **Topic-based publish-subscribe**: In topic-based publish-subscribe systems, all communication between entities is done through "topics", that is an abstraction similar to a message queue or a channel. Publishers using the "publish()" function to send a message to a given topic, while subscribes used the "subscribe()" function to receive messages from the topic.

- **Content-based publish-subscribe**. These systems allow more expressiveness than topic-based systems by allowing subscribers to express interest in the messages they will receive based on the actual content of messages. Subscribers provide predicates to filter the messages that they are interested in receiving.

- **Type-based publish-subscribe**. Type-based publish-subscribe systems allow subscribers to specify which messages they want to receive based on the "type" of the message.

In this project, we are going to be considering a topic-based publish-subscribe system because a significant portion of popular and commercially available publish-subscribe systems are topic-based. Due to their simplicity of interest matching, they are easier to implement and are generally more efficient than the other categories.

5.1.2 Data Delivery Guarantees

Applications using publish-subscribe systems typically expect certain functional guarantees on data delivery from publishers to subscribers. The two most common properties that applications require are *exactly-once message delivery* and *message persistence*. This subsection discusses these guarantees, that **ePulsar** aims to provide as well.

Exactly-once Message Delivery

Each data subscriber expects every message that was successfully published (and acknowledged) to be received only once. A message can neither be dropped nor delivered more than once. This approach requires that (1) each message has a unique identifier that distinguishes it from other messages; (2) publishers are able to re-send messages that were not successfully published; (3) brokers are able to perform de-duplication of the same message that was published multiple times; and (4) subscribers track the last message that they received, and can query the broker for any newer messages that it might have for the given subscriber.

Message Persistence

A typical distributed system has several communicating entities, all of which cannot be online at the same time. A publish-subscribe system should be able to tolerate the disconnection of subscribers at a time when publishers are publishing messages. A subscriber client should be able to reconnect at a later point in time and be able to receive the pending messages, which requires that the publish-subscribe middleware not only maintains a log of the messages that were published, but also keeps track of which messages have been successfully delivered to each subscriber so that a subscriber is only sent the unread messages when it reconnects.

5.1.3 Message Delivery Latency Guarantee

In addition to the correctness of sending and receiving messages that are provided by data delivery guarantees, emerging applications, such as drone-swarm coordination and autonomous or assisted driving, require that the time spent from the generation of the message, to it being received by the publish-subscribe middleware, till the time when it is received by all the relevant subscribers should be low and bounded. The message delivery latency is composed of (1) communication latency from the publisher to the publish-subscribe middleware; (2) latency incurred in processing and persisting the message on the middleware; and (3) communication latency from the middleware to the subscribers. In a typical scenario, multiple publishers communicate with multiple subscribers, hence the message delivery latency guarantee has to be satisfied for all possible pairs of publishers and subscribers.

5.2 Architectural Components of ePulsar

5.2.1 Data Plane Components

Clients

Clients are geo-distributed and each client is a publisher or a subscriber of a topic. A publisher sends messages to the broker hosting its topic, while a subscriber receives messages published to its topic as notifications.

Brokers

Brokers are the components of any publish-subscribe system responsible for message exchange between producers and consumers. In **ePulsar** , brokers are deployed on geo-distributed Edge sites. Brokers are also associated with another software component called BookKeeper, that acts as the high-performance storage layer. BookKeeper nodes, called *bookies*, persist messages for each topic so that older messages can be retrieved in the event of a broker failure or broker migration.

5.2.2 Control Plane Components

Metrics Store

The Metrics Store receives monitoring metrics from clients and brokers pertaining to resource consumption of brokers, message rates on topics and network proximity between clients and brokers. This monitoring data is accessed by the control-plane for making dynamic topic placement decisions through the broker selection and violation detection policies, that are described next.

Broker Selection Policy

The broker selection policy of **ePulsar** assigns topics to brokers in a latency-aware manner such that the end-to-end latency of data delivery from the producer to broker and finally to the consumer is bounded by the latency threshold provided for the topic. As shown in

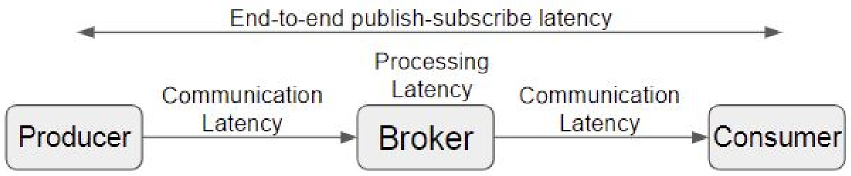


Figure 5.1: Breakdown of end-to-end message delivery latency for a producer-consumer pair.

Section 5.2.2, the end-to-end message delivery latency is made up of the communication latency between the producer and serving broker, the latency of handling the message on the broker (potentially persisting it on durable storage in the case of a persistence topic), and the communication latency from the broker to the consumer. The sum of all these component latencies needs to be below the latency threshold. This constraint needs to be satisfied for each producer-consumer pair of the given topic, that may have disparate connectivity to the serving broker compared to other producer-consumer pairs. Hence the broker selection policy needs to select a broker for hosting a topic that is (1) in network proximity to the clients of the topic to ensure low communication latency; and (2) is not overloaded to ensure low processing latency.

Violation Detection Policy

The violation Detection Policy of **ePulsar** is used to check whether an existing topic in the publish-subscribe system has its end-to-end message delivery latency constraint violated. This computation is done for each topic independently. Since publisher and subscriber clients of a topic can be connected at different points in the network topology, the worst-case message delivery latency is incurred for the publisher-subscriber pair that are the furthest from the broker. Thus, all possible publisher-subscriber pairs have to be analyzed to check for violations.

The worst-case message delivery latency for a topic is the sum of the maximum network latency from any publisher to the broker, the maximum network latency from the broker to any subscriber, and the per-message processing latency on the broker. The worst-case message delivery latency of a topic is compared against the topic-specific threshold, and if the observed latency exceeds the threshold, a violation is detected, and the control-plane triggers the migration of the topic to another broker that can satisfy the latency constraint.

5.2.3 Use of Proposed Mechanisms in Control Plane

ePulsar 's control plane utilizes the Network Proximity Estimation mechanism to estimate the worst-case message delivery latency for a given topic. This mechanism is able to estimate the communication component of the message delivery latency, while **ePulsar** relies on an offline profile of its broker's data plane for estimating the processing latency. This mechanism is used both for (1) determining whether a particular candidate broker will be able to satisfy the worst-case message delivery latency constraint, given the network proximity information between the clients and the candidate broker, and (2) detecting a violation of the worst-case message delivery latency constraint. The clients' network proximity to the current serving broker is used to compute the observed worst-case message delivery latency, that is then compared against the topic-specific threshold.

5.3 End-to-End Latency Estimation using Network Proximity Mechanism

5.3.1 Deployment of Network Coordinate Agents

Estimating the network latency between every client and broker pair requires that all of those entities have an associated network coordinate, such that by computing the Euclidean distance between the coordinates of two nodes, one can compute the communication latency between them. **ePulsar** co-locates one instance of the Network Coordinate Agent with each broker, such that the network coordinate of the agent represents the network proximity information of the broker. Clients that are not mobile (e.g., connected to a wired network) also run a Network Coordinate Agent instance and use the agent's coordinate as their own. To handle mobile clients, **ePulsar** relies on a network of Edge Gateways that each run a Network Coordinate Agent. Each client discovers the current serving Edge Gateway using a DNS-like discovery mechanism, records the network round-trip time to the Gateway and computes its own network coordinate from the network coordinate of the Gateway and the measured RTT to Gateway. The deployment is illustrated in Section 5.3.1.

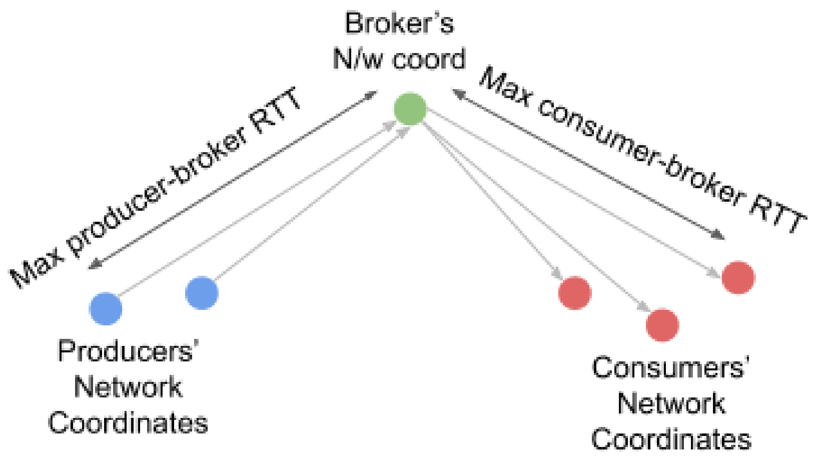


Figure 5.2: Illustration of the use of network coordinates for computing the worst-case message delivery latency for a given topic. Network coordinates can be used to compute the communication component of the message delivery latency. Since all publisher-subscriber pairs communicate via the broker, computing the maximum network latency from the publishers to the broker and that from the broker to the subscribers suffices to compute the worst-case communication latency across all publisher-subscribe pairs.

The network coordinates of all these system entities is utilized by the broker selection and violation detection policies for estimating the communication component of the end-to-end message delivery latency.

5.3.2 Broker Selection Policy

The set of publishers and subscribers for a topic t are denoted by $P(t)$ and $C(t)$ respectively. The worst-case message delivery latency for topic t when served by broker b is represented as shown in Eq. (5.1).

$$W(t,b) = \max_{p \in P(t)} d\left(NC(p), NC(b)\right)/2 + \max_{c \in C(t)} d\left(NC(c), NC(b)\right)/2 + PROC(t) \tag{5.1}$$

Eq. (5.1) is used by the broker selection policy to estimate the worse-case message delivery latency that a given broker can provide for a given topic. The broker selection policy is shown in Algorithm 4. For each topic that needs to be placed on a broker, the policy finds the set of candidate brokers that can satisfy the message delivery latency constraint (line 3-4). Next, the policy ranks the topics in increasing order of the number of possible candidates for each topic and starts broker selection in that order (lines 5-6). The aim of the policy is to successfully place as many topics as possible. Next, if a particular broker

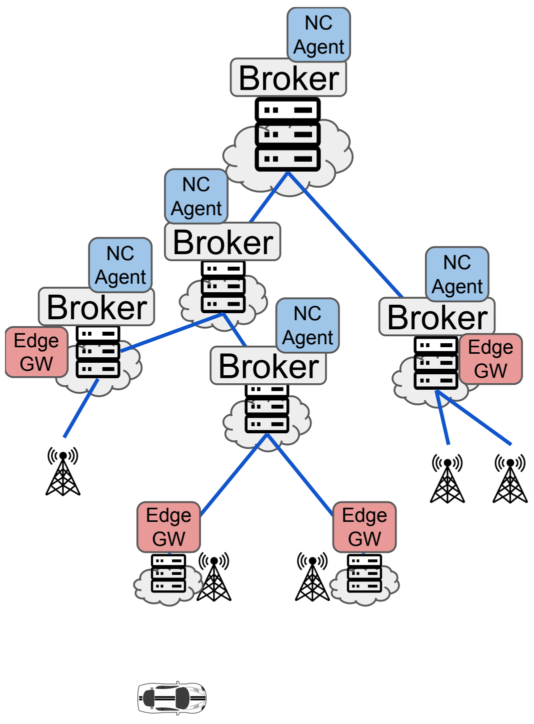


Figure 5.3: Illustration of the deployment of network coordinate agents in **ePulsar** .

can host a given topic without being overloaded, the topic is assigned to that broker (lines 7-9).

5.3.3 Violation Detection Policy

The Violation Detection Policy is invoked periodically, that iterates over all the topics currently in the system and checks if any one of them is undergoing latency violation (lines 4-5). If a topic is experiencing end-to-end latency violation, it is added to the set of topics that need to be migrated to a better broker (line 6). Next the policy checks if any broker is overloaded due to too many topics (line 7). In case there is a broker that is overloaded, the policy fetches the topics that are best candidates for migrating out of the overloaded broker (line 9), and adds it to the set of topics to be migrated. Then the policy calls the topic-placement module that uses Algorithm 4 for selecting the target broker for hosting each topic (line 11). If the chosen target broker is not the same as the current broker, a topic migration is triggered (lines 14-16).

Algorithm 4 Broker selection policy algorithm. Inputs are T (set of topics to place on brokers), current topic-to-broker mapping B, and M (monitoring data)

```
1: procedure SELECTBROKER($T, B, M$)
2:     $F \leftarrow dict\,()$
3:     for $t \in T$ do
4:         $F\,[t] \leftarrow get\_feasible\_brokers\_by\_latency\,(t, M)$
5:     sort $T$ by increasing $|F\,[t]\,|$
6:     for $t \in T$ do
7:         for $b_{cand} \in F\,[t]$ do
8:             if $can\_host\,(b_{cand}, t, B, M)$ then          ▷ no broker overload
9:                 $B\,[t] \leftarrow b_{cand}$
10:                break
11:    return $P$
```

Algorithm 5 Violation Detection Policy algorithm. Inputs are B_0 (initial topic partitioning) and M (monitoring data)

```
1: procedure PERFORMREPARTITIONING($B_0$, $M$)
2:     $R \leftarrow \{\}$                                   ▷ set of migration commands
3:     $T_{migr} \leftarrow \{\}$                            ▷ topics to migrate from curr broker
4:     for $t \in B_0.topics$ do
5:         if $latency\_violated\,(t)$ then
6:             $T_{migr} \leftarrow T_{migr} \cup \{t\}$
7:     $B_{overload} \leftarrow get\_overloaded\_brokers\,(B_0, M)$
8:     for $b \in B_{overload}$ do                           ▷ resource-based detection
9:         $T_b \leftarrow get\_topics\_to\_migrate\,(b, B_0, M)$
10:        $T_{migr} \leftarrow T_{migr} \cup T_b$
11:    $B \leftarrow PlaceTopics\,(T_{migr}, B_0, M)$
12:    $R \leftarrow \{\}$                                   ▷ set of re-partitioning commands
13:    for $t \in B_0.topics$ do
14:        if $B\,[t] \neq B_0\,[t]$ then
15:            $R \leftarrow R \cup \{(t, B_0\,[t]\,, B\,[t])\}$    ▷ add previous broker and new broker
16:    $execute\_reconfigs\,(R)$
```

Table 5.1: Notations used.

Notation	Definition
t	topic
$P(t)$	producers of topic t
$C(t)$	consumers of topic t
$L_{th}(t)$	end-to-end latency constraint for topic t
$B[t]$	broker hosting topic t
$NC(i)$	network coordinate of entity i
$\overline{NC}(I)$	centroid network coordinate of entities in I
$d(nc_1, nc_2)$	distance between network coordinates
$W(t)$	Deviation of client NC from centroids of topic t
$W(t)$	Deviation of client NC from centroids of topic t
$E(t)$	Worst-case end-to-end latency for topic t
$PROC(t)$	Processing latency at broker for topic t

5.4 Distributed Monitoring in ePulsar

5.4.1 Distribution of Monitoring Components over Infrastructure

Each client in **ePulsar** hosts a Metric Agent that monitors the client's network coordinate. Each broker hosts a Metric Agent that monitors its network coordinate and the message rate for each topic. These metrics agents perform Per-Metric Aggregation of each metric with a bucket size of 5 seconds, computing the average over all the measurements in each bucket.

Each broker hosts the Metrics Server along with it. For each topic that is hosted on a given broker, all clients of that topic report their metric aggregates to the Metrics Server hosted on that broker, that performs an aggregation of the network coordinates as discussed in Section 5.4.2. The aggregated metrics are then reported to the control plane's Metric Store to be used by violation detection and broker selection policies.

5.4.2 Monitoring Data Aggregation Logic

ePulsar reduces the amount of per-topic monitoring data sent to the Metric Store by aggregating the network coordinates of multiple clients of each topic. For each topic t that is hosted on broker b, **ePulsar** reports the following aggregate statistics to the Metric Store. Table 5.1 provides a summary of notations used.

- **Producer and Consumer Centroid.** The centroids of producers' and consumers' network coordinates provide an approximate network *location* of a topic's clients.

We compute the centroid producer coordinate $\overline{NC_P}(t)$ and centroid consumer coordinate $\overline{NC_C}(t)$ as follows.

$$\overline{NC_P}(t) = \overline{NC}(\{i : i \in P(t)\})$$

$$\overline{NC_C}(t) = \overline{NC}(\{i : i \in C(t)\})$$

- **Maximum Deviation from Centroids.** To account for the loss of fine-grained network coordinate information due to aggregating them as centroids, **ePulsar** sends the maximum deviation of the clients' coordinates from their corresponding centroid. We denote this as $W(t)$ and it is computed as shown below.

$$W(t) = \max_{p \in P(t)} d\left(NC(p), \overline{NC_P}(t)\right)/2 + \max_{c \in C(t)} d\left(NC(c), \overline{NC_C}(t)\right)/2$$

- **Worst-case Communication Latency.** For each topic t, we compute the worst-case communication latency across all producer-consumer pairs. We denote this as $E(t)$ and it is computed as follows.

$$E(t) = \max_{p \in P(t)} d\left(NC(p), NC(B[t])\right)/2 + \max_{c \in C(t)} d\left(NC(c), NC(B[t])\right)/2$$

  This information is used to determine whether the current broker, denoted by $B[t]$ is meeting the topic's end-to-end pub-sub latency threshold.

One key point to take note of is that the volume of per-topic monitoring data generated is independent of the number of clients using that topic. Given the high heterogeneity of broker and client network locations in a geo-distributed setting, the above aggregation technique significantly reduces monitoring data traffic. By contrast, a naive approach that records the network coordinates of all clients would incur network traffic proportional to the number of clients of each topic. However, the monitoring data used by **ePulsar** 's broker selection policy does not contain individual client network coordinates, but rather the centroids of producer and consumer clients' network coordinates. Hence, Eq. (5.1) in the Broker Selection Policy needs to be updated to use centroid coordinates as shown in Eq. (5.2).

$$W(t,b) = d\left(\overline{NC_P}(t), NC(b)\right) + d\left(\overline{NC_C}(t), NC(b)\right) + W(t) + PROC(t) \quad (5.2)$$

5.5 Implementation of ePulsar

ePulsar is implemented by extending Apache Pulsar v2.2.1. We chose Pulsar as the base system because of the strong data-plane semantics that it offers and also because topic migrations in Pulsar are much more agile than in Apache Kafka. However, Pulsar, like Kafka, is designed for datacenters and bundles topics together for monitoring and placement on brokers. This bundling is done for reducing the amount of metadata that it has to maintain. **ePulsar** needs to independently manage each topic, and hence, we restrict the maximum number of topics in a bundle to be 1. When more than one topics are mapped to a bundle, it results in the bundle being split into two new bundles, each with one topic. By inheriting the concept of bundles from Pulsar, **ePulsar** is able to re-use the bundle-centric monitoring and broker selection in Pulsar, while being able to manage each topic independently.

The control plane policies of broker selection and violation detection are implemented in the Load Manager module of Pulsar. We replace the off-the-shelf Load Manager of Pulsar, that aims at even distribution of workload and prevention of hotspots, with an edge-centric implementation that takes communication latency into account as well. **ePulsar** inherits from Pulsar the use of a ZooKeeper instance as the Metrics Store, that is also used for storing configuration information about the system, e.g., topic ownership of each broker, cluster membership, etc. The monitoring data is queried by the Load Manager module periodically every 5 seconds to detect violations and compute better brokers for topics that are undergoing latency constraint violation.

Brokers in **ePulsar** run on Edge sites along with cloud datacenters. The control plane component of Pulsar, i.e., Load Manager, runs on one of the brokers at a time, that serves as the Leader broker of the cluster. In **ePulsar** the Leader broker is forced to be one that is deployed in a datacenter for better availability. Hence, **ePulsar** has a centralized control-plane located in a cloud datacenter. The ZooKeeper instance that serves as the Metrics Store is co-located in the same datacenter as the Leader broker running the Load Manager module. Clients connect to the publish-subscribe infrastructure via various access media, e.g., cellular (4G LTE), WiFi or wired networks. All entities in the publish-subscribe system, including brokers and clients periodically (every 5 seconds) query their network coordinate, either from the Network Coordinate Agent co-located with them (in the case of brokers and static clients) or from the serving Edge Gateway (in the case of mobile clients) and report it to the monitoring mechanism.

5.6 Evaluations

In this section, we present a set of evaluations to demonstrate the efficacy of the proposed mechanisms in enabling an edge-centric latency-aware control plane for a geo-distributed publish-subscribe system **ePulsar** . Specifically, we aim to prove the following hypotheses through experimentation.

1. Incorporating network proximity information in the broker selection policy allows the satisfaction of per-topic latency constraints. Aggregation of client network coordinates does not impair the quality of the broker selection decision.
2. Distributed monitoring results in monitoring overhead reduction.
3. **ePulsar** is able to meet the end-to-end latency constraints for exemplar applications.

We verify the above hypotheses using two main methods: (1) microbenchmarks that analyze different components of **ePulsar** 's architecture in isolation; (2) end-to-end evaluations of the UAV Swarm application scenario consisting of realistic infrastructure topology and client workload.

5.6.1 Evaluation Scenario

We evaluate **ePulsar** under realistic infrastructure and subscription patterns. We consider the following evaluation scenario from which we design microbenchmarks and end-to-end experiments.

A UAV swarm consists of multiple drones that move together to accomplish a given task. The swarm contains a leader UAV and the rest of the UAVs are followers. Each swarm follows a Random Waypoint mobility model [46].

Subscription Pattern. The leader UAV sends movement commands to the followers through a topic called *follow_leader*. The followers communicate information extracted from onboard sensors to the leader via a topic *sensor_data*. The E2E pub-sub latency constraint is set at 40 ms.

Infrastructure. We consider a city-wide cellular network equipped with Edge resources, where UAVs use 4G-LTE as the communication medium. We assume that the city is divided into multiple *Mobile Edge Computing (MEC)* zones, each with a single edge site. The locations of the edge sites is determined via k-means clustering of the cell tower locations [47] of Atlanta [14]. The Edge sites communicate with each other via a city-level switch, with inter-site RTT of 30 ms. Each Edge site hosts a broker and an Edge Gateway. Each client is directly connected to the Edge site corresponding to its current location based on k-means clustering. Since clients are mobile, they query NC from the Gateway running on the respective sites they are directly connected to. The broker running the Load Manager component and the ZooKeeper instance is hosted in the cloud with a one-way latency of 40 ms to any Edge site.

5.6.2 Evaluation Platform

The evaluation scenario described in Section 5.6.1 poses the following requirements to be satisfied by the evaluation platform: (1) support a heterogeneous network topology; (2) allow emulation of unmodified software components (**ePulsar** entities and clients); and (3) emulate device mobility. To satisfy these requirements, we use the *Containernet* [48] evaluation platform, that has also been used by previous Edge computing research [49, 50]. Containernet uses Docker containers as hosts (allowing the use of unmodified software entities) in network topologies emulated using Open vSwitch. We set custom latencies on the network links using the Linux tool *tc* (to support heterogeneous topologies), and remove/create network links on the fly (to emulate device mobility). The emulated infrastructure is deployed on an Ubuntu 16.04 VM with 48 CPU cores and 64 GB RAM. We use Docker's resource reservation to allocate dedicated resources to each container and minimize performance interference.

5.6.3 Evaluation of Broker Selection Policy

In this section, we evaluate the effectiveness of **ePulsar** 's broker selection policy to meet E2E message delivery latency guarantees for realistic infrastructure topologies and client subscription patterns. We compare the proposed policy against the following two baselines:

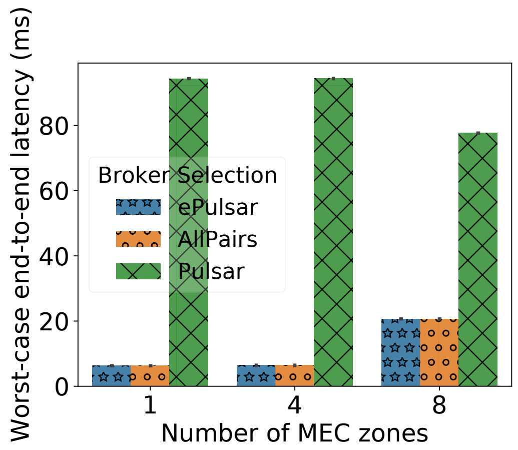
- **AllPairs**. Same as **ePulsar** , but instead of clients' NC centroids, **AllPairs** uses the NC of each individual client to compute the expected E2E latency for each producer-consumer pair. A broker is chosen only if the worst-case E2E latency falls below the threshold.

- **Pulsar**. Apache Pulsar offers well-developed data-plane semantics that are appropriate for the target applications for the edge. Therefore, we choose Pulsar as the other baseline. Pulsar uses consistent hashing to compute the hash for a topic name. The output space of the hash function is divided among all brokers uniformly. The topic is assigned to the broker whose hash-space partition contains the topic's hash.

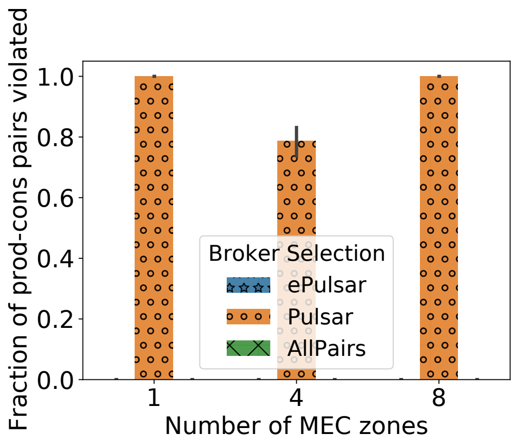
For the representative application scenario mentioned in Section 5.6.1, we generate infrastructure topologies with varying degrees of geo-distribution by varying the number of MEC zones in the metropolitan area. For each such topology, we first emulate the infrastructure of the given topology using Containernet. After allowing the NC agents in brokers and clients to stabilize for 10 minutes, we query each agent's coordinate. The querying is done once per topology. Using the coordinates of all nodes in the topology, we can then estimate the E2E delay for any producer-consumer pair of a topic given the location of the clients and the broker hosting that topic. Based on the application scenario's subscription pattern, we determine the clients for each topic and place them on the nodes of the generated topology.

The network coordinates of the producers and consumers for each topic serve as the input to the broker selection policy. We analyze the result of the policy in terms of the E2E latency and violation ratio. Violation ratio represents the fraction of producer-consumer pairs for whom the latency threshold is violated. In the experiments, we consider 1000 different random permutations of client placement and topic subscriptions. The intent is to have a large coverage of possibilities wherein clients could be located in different geographical areas and/or could be subscribing to different sets of topics.

We vary the number of MEC zones in the simulated metro area and distribute 16 UAV swarms in the city. Each swarm comprises 8 UAVs, with one of them serving as the leader. UAVs follow the subscription pattern described in Section 5.6.1. Figs. 5.4a and 5.4b show



(a) Worst-case E2E latency



(b) Violation Ratio.

Figure 5.4: Analysis of broker selection policy for UAV swarm application scenario.

the worst-case E2E latency and violation ratio over all producer-consumer pairs. Since **ePulsar** performs latency-aware broker selection, the worst-case latency remains under the threshold, resulting in no violations even when the number of MEC zones is increased. The results in this subsection validate the first hypothesis that the incorporation of network coordinates information in the broker selection policy improves the satisfaction of end-to-end message delivery latency constraint. Furthermore, the comparison with the AllPairs policy shows that the aggregation of clients' network coordinates as centroids does not result in poor broker selection with respect to meeting E2E latency constraints.

5.6.4 Evaluation of Distributed Monitoring

We evaluate the savings in monitoring traffic by aggregating per-topic client NCs at the serving broker before reporting them to the Metrics Store. This traffic is sent continuously through the WAN and impacts the scalability of the system, hence we consider aggregate monitoring traffic rate as the metric-of-interest. We focus our evaluation on a single broker hosting topics with multiple clients - as the behavior is independent of other brokers. We vary the number of topics hosted on the broker and the number of clients connected to each topic.

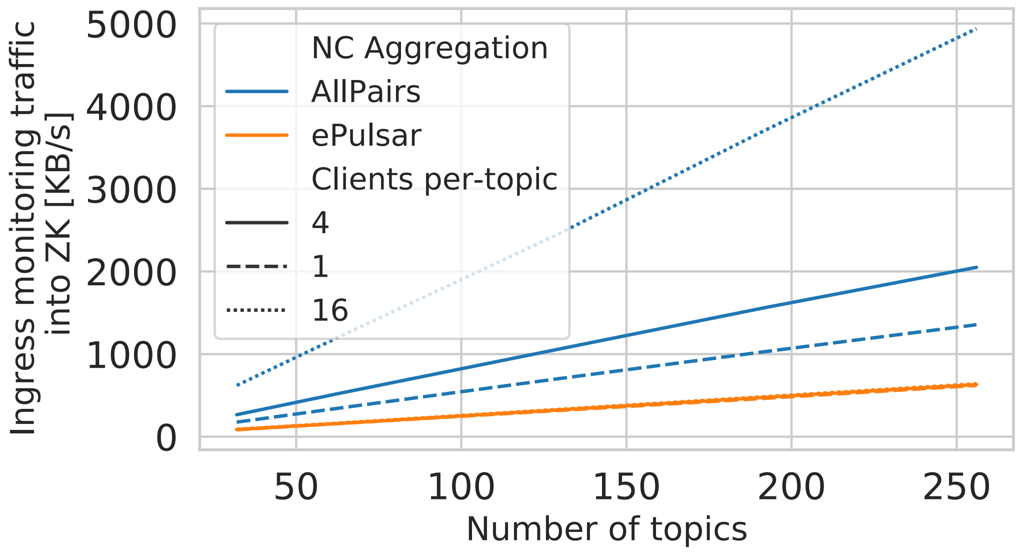


Figure 5.5: Monitoring traffic rate under varying number of topics and clients per topic. **ePulsar** 's NC aggregation results in considerable savings over naive AllPairs.

Fig. 5.5 shows the data rate of monitoring traffic sent to the Metrics Store. An increasing number of topics results in higher data rate. The rate of increase is higher without centroid aggregation (AllPairs policy) and also with more clients per topic. **ePulsar** 's aggregation, however, causes data rate to be independent of the number of clients - since a constant

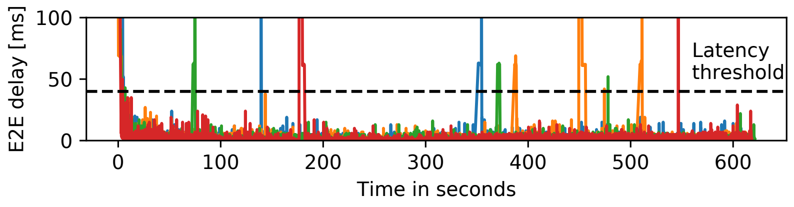


Figure 5.6: E2E latencies experienced by the 8 independent drone swarms for their respective representative topic over time. Latency violations (transient spikes) are observed when a swarm moves from one MEC zone to another. For a brief period, the latency remains higher than the threshold due to network traversal through the city-level switch. **ePulsar** 's topic migration brings the latency back under the threshold.

amount of data is sent to the Metrics Store per topic. These results validate the second hypothesis that aggregating clients' network coordinates as centroids results in reducing the monitoring overhead.

5.6.5 End-to-End Evaluation

In this section, we evaluate **ePulsar** 's ability to respect E2E latency constraints of the exemplar UAV Swarm application, and validate the third hypothesis. The metric we use for the evaluation is E2E message delivery latency – i.e., the elapsed time between a client publishing a message to a topic and the receipt of the published message by all the clients subscribing to that topic.

We consider the infrastructure topology described in Section 5.6.1 with 4 MEC zones and emulate it using Containernet. We emulate each UAV swarm as an independent container where the mobility of all the members of the swarm are identical. In the emulated network topology, each zone consists of a network switch to which the broker and the Edge Gateway connect. We create a link between the swarm's container and the switch corresponding to the swarm's current MEC zone. When a swarm moves into a new zone, the link to the previous zone's switch is removed and a link to the new zone's switch is created. We emulate 8 independent swarms, each with 8 UAVs, following the Random Waypoint mobility model in the city at a relatively high speed of 50 meters/sec[1]. Both leader and followers generate 200 msgs/sec each of size 1 KB [51]. We perform this experiment for 10 minutes.

We show the E2E latency of a single representative topic from each swarm in Fig. 5.6[2].

[1]We use such a high speed to trigger several mobility-driven topic migrations during the experiment.

[2]To avoid cluttering the figure, we do not show all the topics of each swarm since their behavior is identical.

For each swarm. the E2E latency remains under the latency threshold (40 ms) for most of the experiment duration. Transient violations of latency threshold occur when a swarm moves into a different MEC zone than the one currently hosting the swarm's topics. **ePulsar** 's monitoring module detects such violations and triggers migration of the swarm's topics to the broker at the new MEC zone, after which the E2E latency returns back under the latency threshold.

5.7 Related Work

There has been prior work in building edge-centric pub-sub systems, which include EMMA [32], FogMQ [52], and MutiPub [53]. However, these systems do not meet the data communication and/or the scalability needs of the aforementioned applications. EMMA does not handle message reliability guarantees or at-least-once/exactly-once semantics that are typically offered by commercial pub-sub systems. FogMQ relies on creating a clone in the proximity of each device to handle communication on behalf of that device. With a large number of participating clients, this design decision will be a huge resource burden on the already scarce edge resources making the system non-scalable. MultiPub aims to provide latency guarantees for multi-region pub-sub systems by relying on having detailed information of inter-region latencies, as well as the network latency between every client-broker pair. Although this might be tractable for deployments with a handful of cloud regions, the much denser distribution of edge sites makes monitoring and maintaining such fine-grained latency information infeasible.

5.8 Conclusion

In conclusion, **ePulsar** is an Edge-centric publish-subscribe system with a control-plane designed to ensure satisfaction of end-to-end message delivery latency constraints for each topic. The key control plane policy of **ePulsar** is selecting a broker for topic placement from a set of geo-distributed brokers. **ePulsar** leverages the network proximity estimation mechanism to select the broker that can satisfy message delivery latency requirement for each topic. It uses the distributed end-to-end monitoring mechanism to detect when a topic's latency threshold is violated and accordingly triggers topic migration. Through evaluations we show that **ePulsar** is able to satisfy the requirements of realistic situation-awareness applications on a realistic Edge infrastructure topology.

CHAPTER 6
ONEEDGE: APPLICATION ORCHESTRATION OVER GEO-DISTRIBUTED EDGE INFRASTRUCTURE

Situation-awareness applications consist of multiple components that need to be deployed on Edge sites to be able to serve clients. The Edge infrastructure is heterogeneous both in terms of network topology and site capacity. Situation-awareness applications require that clients are served by application components that are deployed in network proximity from them so that the response-time requirement of the sense-process-actuate control-loops can be met. Secondly, applications that involve collaboration between multiple clients require that the clients close to each other are served by the same application instance. Finally, continuous client mobility necessitates this mapping from client to application instances to be dynamic, because mobility causes the location and network connectivity of clients to change. Thus having developers be responsible for selecting the right Edge sites for hosting application components and adapting the placement due to client mobility is a non-trivial burden on the developers. Instead, developers expect an application orchestration service to manage applications, such as Kubernetes.

However, the control-plane policies in cloud-based orchestrators such as Kubernetes are insufficient to cater to these requirements. For one, Kubernetes uses a tag-matching based approach for selecting candidate nodes for hosting an application, that is insufficient for capturing the heterogeneity in network latency in Edge infrastructure or for location-aware placement. Secondly, Kubernetes does not monitor observed latencies in the application or the client location and therefore cannot detect the need to dynamically change the application instance serving the client.

To overcome these limitations, we propose **OneEdge** – an application orchestrator designed for serving situation-awareness applications on Edge infrastructure. **OneEdge** performs latency-sensitive placement of application components so that the latency requirements of applications are met. It also performs mapping of clients to application instances in a way that satisfies both latency and spatial affinity. Furthermore, it continuously monitors observed latencies and client location to detect any latency or spatial affinity violations and triggers an update in the client to application instance mapping if necessary. More concretely, **OneEdge** makes the following contributions:

- **OneEdge** leverages the Network Proximity Estimation mechanism for performing latency-sensitive application placement over Edge infrastructure.

- **OneEdge** leverages the Dynamic Spatial Context Management mechanism for performing location-aware mapping of clients to application instances.
- It uses the End-to-End Monitoring mechanism to detect if the response-time requirement of an application instance is violated and determines the root-cause of the violation. If the violation is caused by client mobility, it triggers the migration of that particular client to a better application instance. If the violation is caused due to compute overload at a particular application component instance, it triggers a partitioning of the clients served by that instance and moves a subset of clients to another instance. If the application component is one involving inter-client coordination, the client partitioning is done in a location-aware manner by using the Dynamic Spatial Context Management mechanism.

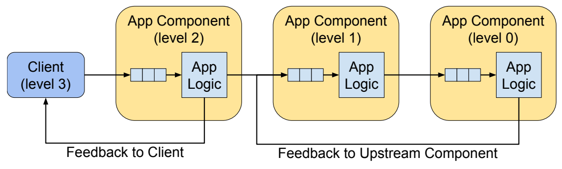
The roadmap of this chapter is as follows. Section 6.1 provides a background of application orchestration on an Edge infrastructure and the application model. It also discusses in detail the requirements of situation-awareness applications and the challenges of meeting those requirements in an Edge setting. Section 6.2 presents the architecture of **OneEdge**. Section 6.3 describes how **OneEdge** utilizes the network proximity estimation mechanism for performing latency-sensitive application placement. Section 6.4 discusses how **OneEdge** uses the Dynamic Spatial Context Management mechanism for location-aware application placement. Section 6.5 describes how **OneEdge** uses the Distributed End-to-End Monitoring mechanism to implement its violation detection and root-cause analysis policies. Section 6.6 presents implementation details and Section 6.7 presents results of experimental evaluation of **OneEdge**. Section 6.9 concludes the chapter.

6.1 Basics of Application Orchestration in Edge Computing

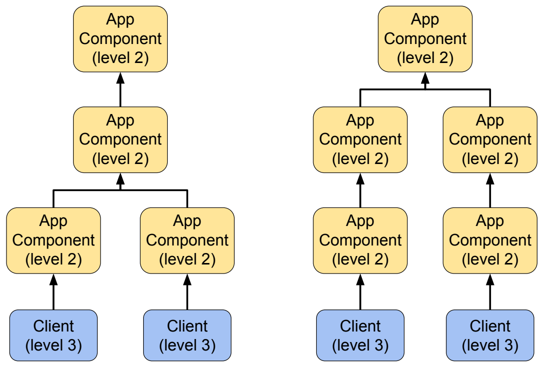
In this section, we will set the stage for the discussion of how **OneEdge** leverages the mechanisms proposed in this dissertation to implement control-plane policies for application orchestration. We first present the application model that **OneEdge** supports, the application requirements for which control-plane policies need to be designed, and the challenges faced by in an Edge setting due to the dynamism in workload.

6.1.1 Application Model

Situation-awareness applications process data incoming from sensors through a series of functions, each extracting out certain information from the input data or performing a certain operation. This can be naturally modeled as a Data Flow Graph (DFG) [54]. Each node in a DFG represents a processing function and each directed edge represents a data dependency between the upstream and downstream node. In **OneEdge**, we assume a special case of the more general DFG model, that is a pipeline of functions. As shown in Fig. 6.1a, each DFG node, or application component, is an independent actor, and is represented by a level. Level 0 is assigned to the most downstream component, and the level number increases as we go upstream toward the client. Each component reads from a queue of input events that is populated by the upstream component and sends output events to the downstream component. An application component is also able to send messages to an upstream component (including the client).



(a) Application Pipeline.



(b) Actual deployment of a pipeline-based application model resembles a forest with multiple trees.

Figure 6.1: Description of the application model.

Although applications are modeled and specified as a linear pipeline, upon deployment for multiple clients, the set of application components and the data dependencies among them resemble a forest, as shown in Fig. 6.1b. This is because in order to serve many geo-distributed clients, the same application component needs to have several instances, which we call *workers*, deployed in network proximity of clients so that communication latency to the application instance can be minimized and real-time response made possible. However, not all pipeline components have stringent latency requirements, and thus can serve multiple clients. Each tree in the forest has the most downstream application component (with level 0) as the root node and clients as leaf nodes. Each root-to-leaf path in a tree is a complete application pipeline, and we call each such path except the leaf node an *application instance*. An application component that serves more than one upstream components is essentially processing information gathered from multiple clients, and therefore is able to enable state sharing among clients. However, even for applications that don't share state among clients in their logic, sharing one or more components in the application pipeline among multiple clients can be beneficial. The benefit of sharing pipeline components comes from the lower memory footprint of sharing an application component compared to running multiple independent application components to serve the same number of clients.

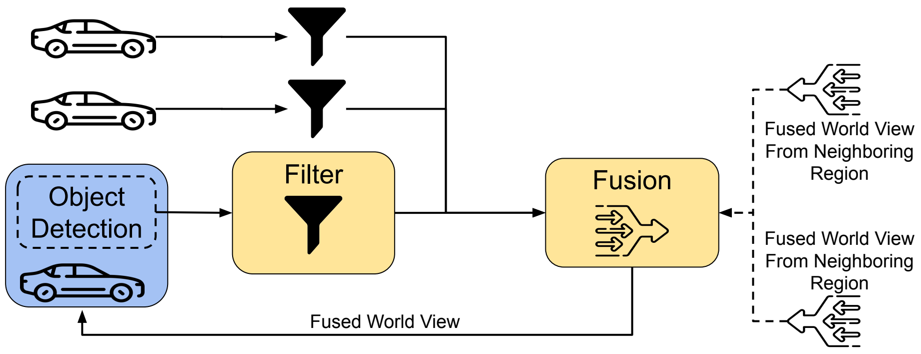

Object Detection
Filter
Fusion
Fused World View From Neighboring Region
Fused World View From Neighboring Region
Fused World View

Figure 6.2: The collaborative driving assistance application modeled as a pipeline of components.

For example, the Collaborative Driving Assistance application can be modeled as a pipeline of application components as shown in Fig. 6.2. The Client component performs object detection on an input LiDAR sensor stream to generate a list of objects that it can see in its immediate field of view. The detected object features are first filtered by a Filter component and then fused with features from other cars in the same region by the Fusion component. The Fusion process aggregates the individual views from multiple vehicles that are in close spatial proximity of one another to create a composite view. This composite view is fed back to the vehicles so that they can improve their lane control and collision avoidance decisions. Each Fusion component instance receives the fused world view from instances responsible for the neighboring regions to incorporate into its own processing. This information is needed to serve clients that are located at region boundaries.

6.1.2 Decisions taken by an Application Orchestrator

Effective orchestration of situation-awareness applications on Edge infrastructure requires two main decisions that need to be taken by the control-plane. These two decisions are (1) mapping clients to application instances; and (2) deploying and managing resources for application instances on Edge sites.

Mapping Clients to Application Instances

We now formally describe the decision concerning mapping clients to application instances for a specific application. Let $\mathcal{C}$ denote the set of clients for the given application and $\mathcal{I}$ denote the set of running instances of all the components of this application. As mentioned before, each application component in $\mathcal{I}$ as well as each client in $\mathcal{C}$ has an associated level, that indicates the stage number in the pipelined application model. The mapping of clients and upstream application components to downstream components is defined by a function $\mathcal{M}$, defined in Eq. (6.1) and Eq. (6.2).

$$\mathcal{M} : (\mathcal{C} \cup \mathcal{I}) \times Z^{+} \rightarrow \mathcal{I} \tag{6.1}$$

$$\mathcal{M}\left(c, l\right) = \text{App Component of level } l \text{ serving } c \tag{6.2}$$

The control-plane needs to compute such a mapping to ensure that the application requirements discussed in Section 6.1.3 are satisfied.

Deployment of Application Instances on Edge Sites

Each application instance, comprising all the components in the application's pipeline model, needs to be deployed on a set of compute resources. The placement of application components ($\mathcal{I}$) on the set of compute resources ($\mathcal{R}$) can be represented by the function $\mathcal{P}$, as described in Eq. (6.3) and Eq. (6.4).

$$\mathcal{P} : \mathcal{I} \rightarrow \mathcal{R} \tag{6.3}$$

$$\mathcal{P}\left(a\right) = \text{Compute resource hosting } a \tag{6.4}$$

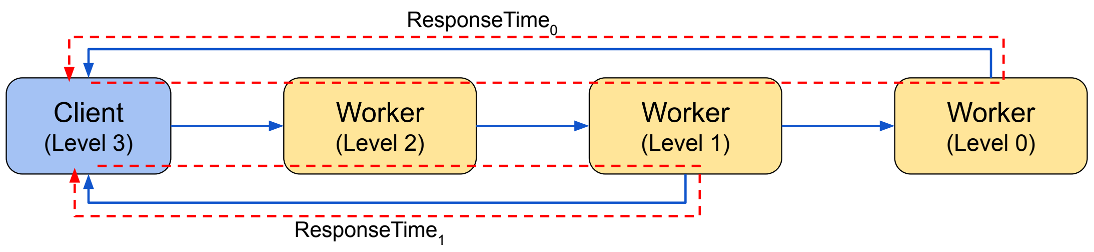


Figure 6.3: A generic pipeline that shows the response time constraints that applications are allowed to specify. The response time of a given application component is the time duration between the client generating a message to the given component processing the message and the result arriving at the client.

The placement of application components is computed in a manner that meets the response time requirements of the application, as described in Section 6.1.3.

6.1.3 Application Requirements

We now discuss the requirements that **OneEdge** allows application developers to specify for an application.

Response Time of Application Component

Application developers are able to specify the response time requirement for each component in the application pipeline model. Response time for an application component is the time elapsed between when a specific data-item/event is generated by the client to when it is processed by the application component and the result is received by the client. This includes the network transmission latency and processing time of all the upstream components and the final network latency between the given component and the client. Let us assume an application with the application model being a pipeline with L stages. The response time constraint for each application component should be satisfied for each client of the given application. The response time for client c from a worker with level l is calculated using total processing latency (defined in Eq. (6.5)), that is calculated recursively using the value for the upstream component. The base case is when the level l corresponds to the level of the client itself, in which case, the total processing latency is equal to the processing latency on the client.

$$total\_proc\,(l) = total\_proc\,(l+1) + N_{(l+1,l)} + C_l \tag{6.5}$$

$$response\_time\left(l\right) = total\_proc\left(l\right) + N_{\left(L-1,l\right)} \tag{6.6}$$

Spatial Affinity

Several situation-awareness applications such as the collaborative driving assistance application (Fig. 6.2) consist of components that are tied to a specific geographical area, and are meant to serve clients located in that geographical area only. This is meant to enable information sharing between clients that are in close spatial proximity to one another. We denote the geographical area served by an application component as its *spatial context*. The partitioning of the geographical space into spatial contexts is application-specific. More precisely, the application developer would specify a function $\mathcal{S}$ that maps a geographical location (x, y) to a unique spatial context s_z (identified by a positive integer z) (Eq. (6.7)).

$$\mathcal{S} : X \times Y \rightarrow \{s_0, s_1, \cdots\} \tag{6.7}$$

$$X = \{x \in R \mid -\pi < x < \pi\} \tag{6.8}$$

$$Y = \{y \in R \mid \frac{-\pi}{2} < y < \frac{\pi}{2}\} \tag{6.9}$$

For an application component of level l that needs to facilitate inter-client information sharing, the control-plane assigns each spatial context s_z with an application component and map all clients located within that spatial context to that instance. Multiple spatial contexts can also be mapped to the same application instance. However, the main requirement is that all clients within given spatial context should be served by the same application instance so that they can effectively share state with other clients in the same spatial context. The quality of this client-application-instance mapping is quantified using the Spatial Alignment metric, as shown in Eq. (6.10). The Spatial Alignment metric is defined for each spatial context s_z that is served by the application component instance a_z at level l. Ideally, the spatial alignment metric should be equal to 1 for all spatial contexts.

$$SA\left(a_z\right) = \frac{\left|\{c \in \mathcal{C} : c.loc \in s_z : \mathcal{M}\left(c, l\right) = a_z\}\right|}{\left|\{c \in \mathcal{C} : c.loc \in s_z\}\right|} \tag{6.10}$$

6.1.4 Application Abstract: Specifications provided by the Developer

OneEdge expects the developer to provide specifications for each application, that should contain information that will be used by the control-plane for making policy decisions. The application abstract should contain the number of levels in the application pipeline, the expected inter-arrival time of sensor data at each client and the following fields for each level of the pipeline.

- The response time constraint for that level, that is used by the application scheduling policy in the control-plane to ensure that the response time observed by all workers that belong to this particular application and level satisfy the constraint. This information is also used by the violation detection policy to detect a violation in response time and trigger a reconfiguration.
- The number of CPU cores and memory (in KB) required by a worker of that level. These are denoted by CPU_{req} and MEM_{req} respectively.
- The expected processing latency of the worker for different number of clients connected to it. This information is obtained by offline-profiling of the application component by the developer. This information is used by the application scheduling policy to estimate the processing latency of a worker.
- Maximum number of clients that can be connected to a worker of that level. This information is obtained by offline profiling of the application component and determining when the worker cannot handle any more clients without a significant increase in processing latency. It is used by the application scheduling policy to select candidate workers for reuse.

6.1.5 Challenges in Achieving Application Objectives

We now discuss the challenges that a system like **OneEdge** would encounter when orchestrating situation-awareness applications on Edge infrastructure.

Heterogeneity in Edge Infrastructure Topology

The network connectivity between Edge Sites that belong to the same geographical area is heterogeneous. Communication latencies between Edge Sites and between clients and Edge Sites depends on how the Edge Sites and clients connect to the network and the peering between the different providers serving these entities. For a particular instance of an application, not all Edge Sites are equally suitable for hosting the different application components. Therefore when selecting the Edge sites for hosting application components, it is necessary that the response-time of all the relevant application components are under their respective thresholds.

Client Mobility

Typical situation-awareness applications have clients that are inherently mobile, for example, vehicles, pedestrians, UAVs, etc. The mobility of clients creates two main challenges. For applications involving inter-client coordination, the mapping of a client to an application component instance is based on the spatial context of that instance and the location of the client. Due to client mobility, the client's location might change so much that it exits the spatial context of the current application component instance, and thus is not able to coordinate with the correct subset of other clients that are in its physical proximity. This requires that the client be migrated to the application component instance that has the correct spatial context that corresponds to the current location of the client. Secondly, client mobility can result in a change in the network routing and hence communication latency to and from the application instance. This change in communication latency affects the response time for that client by the current application instance. This necessitates that the client be migrated to an application instance that can satisfy the response time requirements.

Changes in Processing Requirements of Applications

The frequency of events generated by different application components in an application pipeline changes over time. This is either due to the mobility of clients that changes the properties of the environment sensed by the clients. For instance, in the Collaborative Perception application, the number of neighboring vehicles output by the Detection component (in the client) is a function of the density of neighboring traffic, that changes with time as the ego vehicle moves. The change in processing requirements of applications can also happen for static sensors, such as CCTV cameras, when the environment they are sensing undergoes changes. For instance, in the View-Fuse application, the frequency of events generated by the Detection component depends on the number of cars in the field-of-view of a camera, that changes over time.

6.1.6 Edge-centric Policies for Application Orchestration

The control plane of **OneEdge** contains several edge-centric policies for application orchestration that can satisfy application requirements under the aforementioned challenges. Firstly, **OneEdge**'s application placement policy takes into account response time threshold of application components as well as spatial affinity requirements. Secondly, **OneEdge** performs continuous monitoring of application instances to detect violation of response-time or spatial affinity requirements. In the case of a violation, it performs a root-cause analysis to detect the root-cause of the violation and triggers an appropriate reconfiguration action. The reconfiguration action depends on the nature of the violation. If the violation is due to communication latency increase between an upstream and downstream component, the downstream component instance is mapped to a different instance of the upstream instance so that the latency requirements can be satisfied. If the violation is due to compute overload, a portion of the clients served by the overloaded instance are migrated to a different application instance. If the overloaded instance is spatially constrained, then the partitioning of clients is done so as to ensure that the resulting mapping of clients satisfies spatial affinity requirements.

6.2 Architectural Components of OneEdge

Now we present the system architecture of **OneEdge**, covering the design choices and deployment characteristics of each component of the system. We will first provide a high-level overview of the system, with an emphasis on describing how **OneEdge** operates and the typical operations of each component. Then we will discuss the functions of each component in more detail.

6.2.1 High-level System Architecture

Fig. 6.4 shows the high-level system architecture of **OneEdge**. There are three top-level entities *Clients*, *Edge Sites* and the *Controller*. Client entities represent all client devices, such as autonomous vehicles, drones, etc. that run the client component of situation-awareness applications. Clients are inherently mobile and geo-distributed. The client module of situation-awareness applications hosted by them require connection to an application pipeline that can carry out the rest of the functionality of the application. Each client is also equipped with sensor and actuator devices. Sensors on the client act as the source of data for the application pipeline.

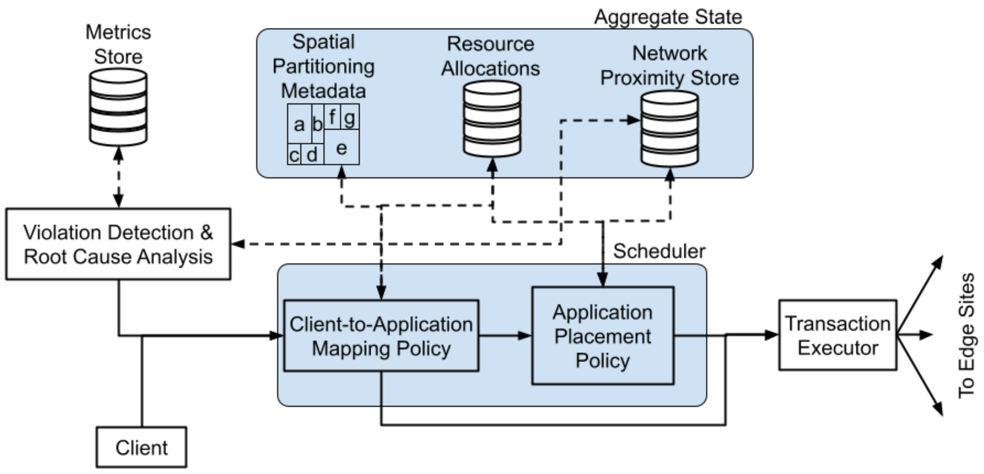

Metrics Store
Aggregate State
Spatial Partitioning Metadata
a b f g
c d e
Resource Allocations
Network Proximity Store
Violation Detection & Root Cause Analysis
Scheduler
Client-to-Application Mapping Policy
Application Placement Policy
Transaction Executor
To Edge Sites
Client

Figure 6.4: **OneEdge**'s System Architecture. A central controller contains the Scheduler, which receives requests either directly from clients or from the monitoring subsystem through violation detection. The Scheduler makes decisions by reading the Aggregate State. Decisions made by the Scheduler are executed on the Edge sites by the Transaction Executor component.

Edge Sites are geo-distributed entities with compute and storage resources that are able to run instances of application components and serve clients. The Controller is a logically-centralized entity that is responsible for performing the main decisions of an application orchestrator, i.e., mapping upstream application workers to downstream workers, and deploying and managing workers (Section 6.1.2). It does so by maintaining the state of the entire infrastructure, including clients, Edge sites and the workers running on Edge sites. The controller receives requests from clients for connection to an application pipeline and it uses the application scheduling policies against the aggregate state to make this decision. In addition, the controller also hosts the policies for detecting violation of application requirements and triggering reconfiguration actions by invoking the aforementioned application scheduling policies.

6.2.2 Client

In the **OneEdge** application orchestration system, a Client consists of three key components - the Client Worker that hosts the client application module, Worker Agent that acts as the interface between the application module and the rest of the system and the Sensor and Actuator devices, that are the source and sink of data.

Client Worker

The Client Worker hosts the application logic that needs to run on client devices. It is responsible for communication with the sensor and actuator devices present on the client. The Client Worker processes data from the sensor and sends the extracted information to the downstream application component. The Client Worker also receives information from the downstream component that it processes to generate commands for the actuator devices.

Client Worker Agent

The Client Worker Agent is an instance of **OneEdge**'s Worker Agent, that acts as the interface between the application logic and the rest of the system. Firstly, the worker agent calls the *on_message_arrival* handler in the client worker whenever a message is received from the sensor. Second, the worker agent is responsible for communicating with the controller to establish a connection with a downstream application component. When the client worker generates a message that needs to be sent to the downstream worker, the worker agent is responsible for sending it to the worker agent of the downstream worker, that then passes it to the corresponding application module. Furthermore, it also serves as a part of the monitoring subsystem, whereby it collects metrics related to the execution of the client worker and forwards them for further processing.

6.2.3 Edge Site

An Edge site consists of three main components, as shown in the right blow-up in Fig. 6.4 - the Site Agent, Container Runtime and Worker Agent.

Site Agent

The Site Agent component is responsible for managing the resources on the site, along with the application component instances running on the site. It interfaces with the central Controller for the scheduling of applications on that Edge site. The Site Agent is responsible for launching application component instances on the Edge Site and allocating resources to those instances on behalf of the controller.

Container Runtime

The Container Runtime is the software platform upon which the various application component instances run as containers. The container runtime provides primitives for launching containers based on an application-specific image, allocation a specific amount of compute and memory resources to containers to ensure predictable performance and isolation, access to a filesystem for storing ephemeral state and communication with other entities in the system.

Worker Agent

The Worker Agent is a software agent that is deployed alongside the application logic within each application component instance's container. It acts as the interface of the application logic for that specific application component with the rest of the components and the outside world. It does so by implementing the various interfaces provided to the application developer. The worker agent facilitates communication between various application instances by allowing upstream and downstream components to send messages to each other, as well as trigger callback functions in the application logic upon message arrival. Furthermore, it also serves as a part of the monitoring subsystem, whereby it collects metrics related to the execution of the application component and forwards them for further processing.

6.2.4 Controller

The Controller is responsible for application scheduling and management at a global-scale across Edge sites. It receives requests for application deployment and reconfiguration from clients and the monitoring subsystem respectively. These requests are processed by the Scheduler, that turns these requests into a transaction (set of actions) to be executed on one or more Edge sites. The transactions are executed on the Edge sites by the Transaction Executor component. We will discuss each of the components in greater detail next.

Scheduler

The Scheduler picks up one request at a time from the Controller's request queue and computes a scheduling decision for the request. The scheduling decision is computed by executing the placement algorithm for mapping the requesting client to a suitable application instance that can satisfy the response time and spatial affinity requirements. If no such application instance exists, a new instance is instantiated and suitable Edge sites for hosting the application components are selected. In the case of a reconfiguration request for updating allocation of an application instance, the scheduler computes the final resource allocation for the components of that application instance.

The Scheduler reads the Aggregate Resource State to check the current available resource capacity on each Edge site, and updates the state with changes to the resource allocation. Hence, the Aggregate State is *optimistically* updated even before the scheduling decision has actually been applied on the specific Edge sites. By doing so, the process of compute application scheduling decisions is decoupled from the actual enforcement of those decisions on the Edge sites. The scheduling decision is in the form of a Transaction, that is a collection of the actions that need to be taken to execute the scheduling decision. The constituent actions of a transaction need to executed on one or more Edge sites.

Aggregate State

The Aggregate State contains information about the Edge Sites, application clients and workers, that is used by the application scheduling policy for making decisions. The Aggregate State contains the following data structures.

- ***sites***. The Aggregate State maintains a dictionary containing all the Edge Sites' information. The key of the dictionary is the ID of the Edge site, while the value is an $EdgeSiteMetadata$ data structure that contains the following fields.
  - The total CPU and memory capacity of the Edge site CPU_{total}, MEM_{total}. CPU_{total} is expressed in number of CPU cores, while MEM_{total} is expressed in kilobytes (KB).
  - The network coordinate of the Edge site, denoted by NC_{site_id}.
- ***clients***. The Aggregate State maintains a dictionary containing information of all the clients present in the system. The $clients$ dictionary maps the ID of a client to an instance of the $ClientMetadata$ data structure, that contains the following fields.
  - The application ID of the client, denoted by app_id.
  - The network coordinate of the client, denoted by NC_{client_id}.
- ***workers***. The Aggregate State maintains information about the application workers currently hosted on edge sites in the $workers$ dictionary. The key of the $workers$ dictionary is the ID of an Edge site, and the value is a lower-level dictionary specific to that Edge site. The lower-level dictionary maps the ID of a worker to an instance of the $WorkerMetadata$ data structure, that contains the following fields describing a particular worker.

  - The ID of the Edge site on which the worker is hosted, denoted by $site\_id$.
  - The number of CPU cores and memory (in KB) allocated to the worker, denoted by CPU_{alloc} and MEM_{alloc} respectively.
  - The application ID and level of the worker, denoted by app_id and $level$ respectively.
  - The number of clients connected to this application worker, denoted by $curr\_clients$.

- ***parent_map***. As discussed in Section 6.1.1, clients and application workers have a parent-child relationship, wherein the parent worker for a child client or worker represents the downstream worker that receives output data from the child client or worker. A worker can have one or more children, but at most one parent. The aggregate state stores this parent child relationship in the form of a dictionary, that maps the ID of the child worker or client to the ID of the parent worker and the ID of the site on which the parent worker is hosted.

As described in Section 6.2.4, the Aggregate State is used by the Scheduler to make application scheduling decisions. For subsequent requests' processing to be aware of the decisions made for the current request, the Aggregate State is updated with the current request's decisions.

Transaction Executor

The Transaction Executor module is responsible for executing the decisions taken by the Scheduler on the Edge Sites. It sends commands to the Site Agents for instantiating workers and allocating resources to them on the target Edge sites.

6.3 Latency-aware Application Scheduling using Network Proximity Estimation

6.3.1 Deployment of Network Coordinate Agents

Network Coordinate agents are deployed on all Edge Sites's Site Manager modules, Edge Gateways and Clients. The agents on the clients don't participate in the decentralized network coordinates protocol, but rather compute their own network coordinate by using the coordinate of their current Edge Gateway. All agents periodically send their current network coordinate to the controller, that maintains them in its aggregate state can be queried by the application scheduling policy.

6.3.2 Estimating Response Time from an Application Worker

Estimating the response time observed from a worker (instance of an application pipeline stage) is equal to the sum of the processing and communication latencies of all the upstream workers including itself and the communication latency from the given worker back to the client.

Estimating the communication latency is done by using the Network Proximity Estimation mechanism. The network latency between the client and its *parent* worker is computed by using the network coordinate of the client and the network coordinate of the Edge Site hosting the worker. Similarly the network latency between a *parent* and a *child* worker is computed using the network coordinates of the two Edge Sites hosting those workers.

The processing latency of a worker is derived from the developer's specification, provided as part of the application abstract. In the context of **OneEdge**, queuing latency at a worker is also included in processing latency. The processing latency of a worker depends on the number of clients it is serving. The developer is expected to perform offline profiling of each application component to generate the distribution of expected processing latencies for a given number of clients served by a worker.

6.3.3 Control-Plane Policy for Latency-aware Application Scheduling

The Latency-aware Application Scheduling policy aims to place applications components on the geo-distributed Edge infrastructure such that response time requirements of each application component is met as well as scarce Edge resources are used only for latency-sensitive application components. A greedy placement policy would place all components of the application on the Edge, that would result in scarcity of resources to support new clients.

OneEdge's latency-sensitive application scheduling policy is illustrated in Algorithm 6. The algorithm jointly performs the mapping of upstream application workers to downstream ones along with the creation of new workers if they don't exist. The algorithm starts with the client worker, and tries to map it to an existing downstream worker or create a new downstream worker, and so on. This process is executed in the $find\_pipeline$ function, that performs a backtracking search to find the right downstream worker to map the current worker to, assuming that rest of the downstream workers will also find a suitable mapping. In every step of this process, the algorithm maintains the total processing latency that has been consumed up to and including the current worker. Line 3 shows that when mapping the client worker to a downstream component, the current cumulative total processing latency is assigned a value equal to the processing latency of the client worker. The function $find\_pipeline$ recursively computes the optimal workers for each component of the application pipeline. The result of the $find\_pipeline$ function contains a list of workers for each downstream application component along with information about whether that worker already existed in the system or if it needs to be created (line 7). The metadata of those workers that need to be created is passed to the Transaction Executor module (line 13).

Algorithm 6 Latency-aware Application Scheduling Policy

```
Require: Client c
Require: Aggregate State $\mathcal{A}$
Require: Application Abstract $A$
 1: $appid \leftarrow A.app\_id$
 2: $L_{client} \leftarrow A.num\_levels - 1$                ▷ Pipeline level of the client
 3: $lat^{total}_{proc} \leftarrow c.proc\_latency()$          ▷ Total processing latency consumed so far
 4: $P \leftarrow find\_pipeline\left(c, c, L_{client} - 1, lat^{total}_{proc}, A, null\right)$
 5: $W \leftarrow []$                                       ▷ Worker launch actions
 6: $child \leftarrow c.id$
 7: for $(w, create) \in P$ do    ▷ Establishing parent-child relationship in aggregate state
 8:     $\mathcal{A}.parent\_map[child] \leftarrow (w.worker\_id, w.site\_id)$
 9:     $\mathcal{A}.workers[w.site\_id][w.worker\_id].curr\_clients + +$
10:     $child \leftarrow w.worker\_id$
11:     if $create == True$ then
12:         $W+ = (w)$              ▷ If $w$ is a new worker, it is scheduled to be launched
13: $launch\_workers\,(W)$
14: return $P$
```

The $find\_pipeline$ function is the central piece of the application scheduling policy. It takes a particular worker as input and the cumulative total processing latency up to that worker and tries to find a mapping for the rest of the downstream application components. It does so recursively by trying to map the input worker to an immediately downstream worker, and then calls the function for this downstream worker. If it cannot find a mapping, it backtracks and tries another downstream worker.

First, the function tries to reuse existing application workers, since each application component can serve more than one client. It gathers the candidates for worker reuse in line 5 (pseudocode for this function is shown in Algorithm 8). For each such reuse candidate, if it can serve an additional client (lines 8-9) and if the response time will be less than the threshold (line 10), this candidate is considered, and a recursive call is made to find a mapping for the remaining components. If the recursive call returns a successful result, then the search is complete and the result is returned, along with the chosen worker (lines 11-13).

If no reuse candidates work, the policy will need to create a new worker. To do so, it selects the site that should hold this worker such that the response time constraint is satisfied. It computes the maximum network latency that can exist between the current worker and the candidate site and from the candidate site back to the client (line 16), that is the difference between the latency threshold of the application component to be created and the sum of the total latency spent so far along with the expected processing latency of the application worker to be created. Any Edge site within this latency from the current worker and client can host the downstream worker. The pseudocode of finding candidate sites is presented in Algorithm 9. If the candidate site has enough free resources and it can satisfy latency requirements, then a new worker is added on the site (line 19) in the aggregate state and a recursive call to the algorithm is made (line 21). If the recursive call fails to find a solution, the newly created worker is deleted and the next candidate site is tried (line 23).

6.4 Spatial-Affinity-aware Application Scheduling using Dynamic Spatial Context Management

An application can have multiple components that are tied to their specific spatial contexts. The constraint imposed by **OneEdge** is that the spatial context of an upstream component should be completely present inside the spatial context of the downstream component. This is done to ensure that the each instance of the downstream component has only one parent instance.

6.4.1 Flexible Partitioning of Geographical Space

For each of the application components that are spatially constrained, the control-plane maintains a KD-Tree based spatial partitioning, as discussed in Section 3.2. The definition of the application abstract is updated so as to include information about which components of an application are spatially constrained. The specification of an application component has a boolean attribute $isSpatiallyConstrained$, that is set to True for components that have a spatial context. **OneEdge** uses the Aggregate State to store the spatial partitioning for components belonging to different applications. It does so by maintaining a dictionary $spatial\_partitionings$, for which the key is a pair of application ID and application component level, and the value is the spatial partitioning for that level of the application pipeline.

Algorithm 7 $find\_pipeline$

```
Require: Child worker/client $child$
Require: Client $client$
Require: Worker level to start search $L_w$
Require: Application abstract $A$
Require: Cumulative processing latency spent so far $lat_{proc}^{total}$
Require: Worker to select forcefully $force\_worker$
 1: if $L_w < 0$ then                                        ▷ Search is complete
 2:     return []
 3: $L \leftarrow A.level\_cfgs[L_w]$
 4: $appid \leftarrow A.app\_id$
 5: $\mathcal{M} \leftarrow \mathcal{A}.client\_mapping$            ▷ Mapping of clients to workers
 6: $reuse\_candidates \leftarrow get\_reuse\_candidates\left(appid, L_w, force\_worker\right)$
 7: for $c \in reuse\_candidates$ do
 8:     if $c.curr\_clients < L.max\_clients$ then
 9:         $lat_{proc} \leftarrow lat_{proc}^{total} + NetworkRTT\left(c, child\right)/2 + proc\_latency\left(L, c.curr\_clients + 1\right)$
10:         $lat_{response} \leftarrow lat_{proc} + NetworkRTT\left(c, client\right)/2$
11:         if $lat_{response} < L.response\_time\_threshold$ then
12:             $P \leftarrow find\_pipeline\left(c, client, L_w - 1, lat_{proc}, A, \mathcal{M}\left(c, L_w - 1\right)\right)$
13:             if $P \neq null$ then
14:                 return $[c.id, c.site\_id, False] + P$
15: if $force\_worker \neq null$ then
16:     return $null$
17: $e2e\_slack \leftarrow L.response\_time\_threshold - lat_{proc}^{total} - proc\_latency\left(L, 1\right)$
18: $C \leftarrow get\_candidate\_sites\left(child, client, e2e\_slack, L.CPU_{req}, L.MEM_{req}\right)$
19: for $site \in C$ do
20:     $w \leftarrow \mathcal{A}.create\_worker\left(site, L\right)$
21:     $lat_{proc} \leftarrow lat_{proc}^{total} + NetworkRTT\left(site, child\right)/2 + proc\_latency\left(L, 1\right)$
22:     $P \leftarrow find\_pipeline\left(w, client, L_w - 1, lat_{proc}, A, null\right)$
23:     if $P == null$ then
24:         $A.delete\_worker\left(w\right)$
25:     else
26:         return $[w.id, site, True] + P$
27: return $null$
```

Algorithm 8 $get\_reuse\_candidates$

```
Require: Application ID $appid$
Require: Worker Level $worker\_level$
Require: Candidate to be forced $force\_candidate$
 1: $C \leftarrow \{\}$
 2: if $force\_candidate \neq null$ then
 3:     $C \leftarrow C \cup \{force\_candidate\}$
 4: else
 5:     for $site\_id \in \mathcal{A}.workers$ do
 6:         for $worker\_id \in \mathcal{A}.workers[site\_id]$ do
 7:             $w \leftarrow \mathcal{A}.workers[site\_id][worker\_id]$
 8:             if $w.appid == appid$ and $w.level == worker\_level$ then
 9:                 $C \leftarrow C \cup \{w\}$
10: return $C$
```

Algorithm 9 $get\_candidate\_sites$

```
Require: Child worker/client $c$
Require: Client $client$
Require: Max network latency from $c$ to candidate site and back to $client$ $nw\_lat_{max}$
Require: Free CPU and memory needed on the candidate site $CPU_{avail}$ and $MEM_{avail}$
 1: $C \leftarrow []$
 2: for $site\_id \in \mathcal{A}.sites$ do
 3:     $(CPU_{avail}, MEM_{avail}) \leftarrow \mathcal{A}.get\_free\_resources\,(site\_id)$
 4:     $nw\_lat \leftarrow NetworkRTT\,(c, site\_id)\,/2 + NetworkRTT\,(client, site\_id)\,/2$
 5:     if $nw\_lat \leq nw\_lat_{max}$ and $CPU_{req} \leq CPU_{avail}$ and $MEM_{req} \leq MEM_{avail}$
    then
 6:         $C \leftarrow C \cup \{(site\_id, nw\_lat)\}$
 7: sort $C$ by $nw\_lat$ in descending order
 8: return $C$
```

The KDTree leaf node associated with a given spatially constrained application component instance is split under two scenarios: (i) when the number of clients mapped to that instance exceeds the maximum number of clients specified by the application developer, and (ii) when **OneEdge**'s Monitoring Subsystem detects a violation of response time for a client mapped to the given instance and the root-cause of the violation is the processing latency on the instance.

6.4.2 System Components for Monitoring Spatial Context

The controller's aggregate state maintains the spatial partitioning information for each application component that is spatially constrained. Each tile in the spatial partitioning can be mapped to a worker, that is intended to serve clients in the geographical area corresponding to the tile. Application scheduling decisions are taken by using the spatial partitioning information to ensure that clients inside a given tile are served by the worker corresponding to that tile only.

Each client's worker agent maintains a cache of the current tile of the spatial partition in which it belongs, along with periodically monitoring its current location. If the current location leaves the geographical area of the tile, the worker agent triggers a migration request to the controller, that then maps the client to a worker that corresponds to the new location. The connection information about the new worker is sent to the client's worker agent, along with the bounding-box of the new tile. The client worker agent replaces the tile in its cache with the new tile. In the event of a tile invalidation due to split or merge operation by the controller, all the worker agents that maintain a cache of the tile that was invalidated are notified of the change, and they refresh their cache.

6.4.3 Control-Plane Policy for Spatial-Affinity-aware Application Scheduling

Application scheduling in a spatial-affinity aware manner is an extension of the latency-sensitive application scheduling presented in Section 6.3.3, by making the following additions. Firstly, when creating a worker on a candidate Edge site, we perform the steps as outlined in Algorithm 10.

Second, the function $get\_reuse\_candidates$ is modified to return the worker corresponding to the tile in which the client is currently located as the only possible candidate.

Third, when the application placement policy makes a deployment decision, it checks the number of clients that are mapped to each instance in the application pipeline. In case for some spatially-constrained worker, the number of clients connected to it exceeds the

Algorithm 10 Creating a Spatially Constrained Worker on a Candidate Edge Site

```
1: $w \leftarrow \mathcal{A}.create\_worker\,(site, L)$
2: if $L.isSpatiallyConstrained$ then
3:     $\mathcal{P} \leftarrow \mathcal{A}.spatial\_partitionings[(appid, worker\_level)]$
4:     $\mathcal{P}.tiles[tile\_id] \leftarrow w$
5:     $\mathcal{P}.workers[w.id] \leftarrow tile\_id$
6: $P \leftarrow find\_pipeline\,(w, worker\_level - 1, e2e, app\_abstract, null)$
7: if $P == null$ then
8:     $A.delete\_worker\,(w)$
9:     delete $\mathcal{P}.tiles[tile\_id]$
10:    delete $\mathcal{P}.workers[w.id]$
11: else
12:    return $[w.id, site, True] + P$
```

Algorithm 11 $get\_reuse\_candidates$: With spatial context included

```
Require: Application ID $appid$
Require: Worker Level $worker\_level$
Require: Level specifications $L$
Require: Client loc $loc$
Require: Candidate to be forced $force\_candidate$
1: $C \leftarrow \{\}$
2: if $L.isSpatiallyConstrained$ then
3:     $\mathcal{P} \leftarrow \mathcal{A}.spatial\_partitionings[(appid, worker\_level)]$
4:     $tile\_id \leftarrow \mathcal{P}.get\_tile\_id\,(loc)$
5:     if $tile\_id \in \mathcal{P}.tiles$ then
6:         if $force\_candidate == null$ or $force\_worker.id == \mathcal{P}.tiles[tile\_id].id$ then
7:             $C \leftarrow C \cup \{\mathcal{P}.tiles[tile\_id]\}$
8: else
9:     Perform regular functionality of $get\_reuse\_candidates$ as in Algorithm 8
10: return $C$
```

maximum number specified in the application abstract, a trigger for splitting the tile is raised. Fourth, the control-plane handles the event of invalidation of a tile by using the algorithm described in Algorithm 12. This event happens whenever a tile is split or merged with a sibling.

Algorithm 12 Handling tile invalidation

```
Require: Spatial Partitioning for that worker's level $\mathcal{P}$
Require: ID of the tile that is to be invalidated $tile\_id$
1: $w \leftarrow \mathcal{P}.tiles[tile\_id]$
2: delete $\mathcal{P}.tiles[tile\_id]$
3: delete $\mathcal{P}.workers[w.id]$
4: $C \leftarrow \mathcal{A}.get\_clients\,(w)$            ▷ Get all clients served by $w$
5: $disconnect\_subtree\,(w)$   ▷ Disconnect all workers from parent in subtree rooted at $w$
6: for $c \in C$ do
7:     Submit deployment request for $c$
```

6.5 Distributed Monitoring in OneEdge

The goal of monitoring in **OneEdge** is to detect violations of performance SLO, that in this case is application response time. The response-time constraint is specified for one or more components of the application pipeline. The objective of **OneEdge** is to detect when any constraint on response-time in an application instance is violated, identify the root-cause of the violation and trigger an appropriate reconfiguration action to alleviate the violation. In this section, we will discuss how **OneEdge** uses the distributed monitoring mechanism proposed in Section 3.4 to achieve the aforementioned objectives.

6.5.1 Definition of Metrics used by **OneEdge**

Each application component instance registers the processing latency metrics with the monitoring subsystem. The processing latency at the given application component instance is measured by the worker agent by monitoring the amount of time taken to process each data-item at the corresponding worker.

```
-   entity_id : <ID of application component instance>
    level : <level in application pipeline of component>
    app_unit: <ID of root worker in the application tree>
    metric: "proc_latency"
```

Each client reports a metric that captures its current network coordinate.

```
-   entity_id : <ID of Client>
    level : <level of client in application pipeline>
    app_unit: <ID of root worker in the application tree>
    metric: "nw_coordinate"
```

Each Edge site also reports a metric to capture its network coordinate.

```
-   entity_id : <ID of Edge Site>
    app_unit: <ID of Edge Site>
    metric: "nw_coordinate"
```

In order to meet the aforementioned objectives of detecting violations of latency requirements and identify root-cause of the violation, it is only necessary to process the metrics belonging to a specific application unit together. In **OneEdge** the application unit is identifier of the root worker, because it is common among the application instances of all the clients connected to it, as shown in Fig. 6.1b.

6.5.2 Deployment of Distributed Monitoring Components

Each worker agent collects measurements of processing latency and network coordinate and records them to a locally running Metrics Agent. Each Site Agent also runs a Metric Agent to record measurements of its network coordinate. The Multi-Metric Aggregation is carried out in a centralized manner, and is co-located with the controller.

6.5.3 Violation Detection Policy

Algorithm 13 presents the violation detection policy used in **OneEdge** to detect the violation of latency SLO for an instance of an application component. The algorithm is invoked periodically at the end of every bucket for every application unit in the system. The ID of the application unit, the abstract of the application and the current bucket are provided as inputs to the algorithm. The algorithm first filters the metrics from the metrics repository that correspond to the given application unit (line 1) and then extract the set of clients among those metrics (line 2). In lines 3-7, the algorithm creates a dictionary called $entities$ that stores the processing latency for each worker that belongs to the given application unit. The rest of the algorithm iterates over the set of clients connected to the given application unit, and checks if any client is facing SLO violation at any downstream worker. For each client, the algorithm iterates through all levels of the application pipeline, while maintaining the cumulative processing latency time (line 18). If the response time at a given level exceeds the threshold, the client is marked as violated along with the level whose latency threshold was exceeded.

It is important to note that for a given client, there could be multiple workers downstream from that client that could be facing SLO violation, however, the given violation detection algorithm only reports the upstream-most worker as violated. This is because fixing the violation of the upstream-most worker will also result in reconfiguring the workers downstream from it.

Algorithm 13 End-to-End Processing Latency Violation Detection policy

```
Require: ID of root worker $ID_{root}$
Require: Application abstract $APP$
Require: Current bucket $b_{curr}$
 1: $MD \leftarrow$ Fetch metrics where AppUnit == $A_{00}$
 2: $C \leftarrow \{m.entity\_id \forall m \in MD : m.level == APP.num\_levels - 1\}$
 3: $E \leftarrow \{\}$        ▷ Dictionary to hold processing latencies for each worker
 4: for $m \in MD$ do
 5:     if $m \notin E$ then
 6:         $E[m.entity\_id] \leftarrow \{\}$
 7:     $E[m.entity\_id][m.metric] \leftarrow m$
 8: $violators \leftarrow \{\}$
 9: for $c \in clients$ do
10:     $child \leftarrow c$
11:     $lat^{total}_{proc} \leftarrow E[child][proc\_latency][b_{curr}]$
12:     $level \leftarrow APP.num\_levels - 2$        ▷ Starting from parent of client
13:     while $level \geq 0$ do        ▷ Iterating over all downstream levels of app pipeline
14:         $w \leftarrow \mathcal{M}(child)$
15:         $L \leftarrow A.level\_cfgs[L_w]$
16:         $NC_{child} \leftarrow$ Network coordinate of $child$
17:         $NC_w \leftarrow$ Network coordinate of $w$
18:         $lat^{total}_{proc} \leftarrow lat^{total}_{proc} + dist(NC_{child}[b_{curr}], NC_w[b_{curr}])/2 + E[w][proc\_latency][b_{curr}]$
19:         $lat_{response} \leftarrow lat^{total}_{proc} + dist(NC_c[b_{curr}], NC_w[b_{curr}])/2$
20:         if $lat_{response} \geq L.response\_time\_threshold$ then
21:             $violators \leftarrow violators \cup \{(c, level)\}$
22:             break
23:         $level \leftarrow level - 1$
    return $violators$
```

6.5.4 Violation Root-Cause Analysis Policy

For each client undergoing latency SLO violation, as detected by the violation detection policy in Algorithm 13, **OneEdge** used the violation root-cause analysis policy (Algorithm 14) to identify the source of performance violation. The algorithm first extracts the component latencies that resulted in the violation for the given client c (lines 4-9). Using the component latencies, it computes the response time observed in the current monitoring window (line 10) - that would be over the latency threshold for level l_{viol}. To determine the root cause of this violation, the rest of the algorithm determines which component latency deviated the most from pre-violation levels to the current window. To do so, it first finds the monitoring window (bucket) in which the client did not face SLO violation (line 12), and computes the response time at that time (line 13). Then it computes the relative change in each component latency with respect to the change in observed response time (lines 14-16). Finally, the component that underwent the most change is returned as the source of the violation.

Once the culprit causing violation for a given client is identified, an appropriate reconfiguration action needs to be triggered. In the following, we discuss the reconfiguration action triggered by **OneEdge** in three types of culprits.

- If the violation is due to communication latency of worker w to its downstream component, then a migration request for w is sent to the control-plane. In response to this request, the control-plane would use the application scheduling policy to place all the clients connected to w to a pipeline instance that can satisfy the latency SLOs.

- If the violation is due to processing latency of worker w that is not spatially constrained, then **OneEdge** tries to migrate a portion of the clients connected to w to another pipeline instance so as to reduce the compute workload on w.

- If the violation is due to processing latency on a spatially constrained worker w, then **OneEdge** triggers a split operation for the tile corresponding to w, resulting in a division of clients connected to w into two different pipeline instances.

6.6 Implementation

All the system components of **OneEdge** have been implemented using C++ and tested on Ubuntu 18.04. The communication between the system components has been implemented using ZMQ, while the serialization and deserialization of messages has been done using

Algorithm 14 Violation Root-Cause Analysis

```
Require: Client undergoing latency SLO violation $c$
Require: Application level with latency SLO violated $l_{viol}$
Require: response time SLO for level $response\_time\_threshold$
Require: Current bucket $b_{curr}$
Require: Bucket size $B$
Require: Entities in client $c$'s application unit $E$
1: $E_c \leftarrow \{\}$    ▷ Component latencies in client $c$'s data-path
2: $level \leftarrow c.level$
3: $w \leftarrow c$
4: while $level \geq l_{viol}$ do    ▷ Collect all component latencies through level $l_{viol}$
5:     $E_c[(w, proc)] \leftarrow E[w][proc]$
6:     if $level > 0$ then
7:         $E_c[(w, parent)] \leftarrow dist\left(NC_w, NC_{\mathcal{M}(w)}\right)/2$
8:         $w \leftarrow \mathcal{M}(w)$
9:     $level \leftarrow level - 1$
10: $E_c[(c, feedback)] \leftarrow (NC_w[b_{curr}], NC_c[b_{curr}])/2$    ▷ NW latency of feedback from violated level
11: $lat^{curr}_{response} \leftarrow \sum_{(w,l) \in E_c} E_c[(w,l)][b_{curr}]$    ▷ Current response time at $l_{viol}$
12: $b_{pre} \leftarrow find\_pre\_violation\_bucket\,(E_c, l_{viol}, response\_time\_threshold, b_{curr}, B)$
13: $lat^{pre}_{response} \leftarrow \sum_{(w,l) \in E_c} E_c[(w,l)][b_{pre}]$    ▷ response time at $l_{viol}$ before violation
14: $change \leftarrow \{\}$    ▷ Dict to store relative change in component latency wrt change in response time
15: for $(w, l) \in E_c$ do
16:     $change[(w,l)] \leftarrow \dfrac{E_c[(w,l)][b_{curr}] - E_c[(w,l)][b_{pre}]}{lat^{curr}_{response} - lat^{pre}_{response}}$
17: $culprit \leftarrow \arg\max_{(w,l)} change[(w,l)]$
18: return $culprit$
```

Algorithm 15 Finding most recent bucket before violation of latency SLO

```
Require: Entities in violated client $c$'s application unit $E$
Require: Application level with response time SLO violated $l_{viol}$
Require: Response time SLO for level $response\_time\_threshold$
Require: Current bucket $b_{curr}$
Require: Bucket size $B$
1: $b \leftarrow b_{curr}$
2: $lat_{response} \leftarrow \sum_{(w,l) \in E_c} E_c[(w,l)][b]$
3: while $lat_{response} > response\_time\_threshold$ do
4:     $b \leftarrow b - B$    ▷ Go to previous bucket
5:     $lat_{response} \leftarrow \sum_{(w,l) \in E_c} E_c[(w,l)][b]$
6: return $b$
```

Google Protobuf library. The Container Runtime used on the Edge sites is Docker. All application workers are instantiated on Edge sites as a Docker container running with an application-specific image. The application-specific image is built on top of a base worker image that packages the Container Agent and its dependencies, such as ZMQ and Protobuf. The application-specific image contains the specific binary of the application logic, that is dynamically linked into the Container Agent. The Controller is implemented as a centralized entity, that maintains aggregate state in a Mongo DB database.

6.7 Evaluations

We evaluate **OneEdge** to test the following hypotheses.

- **OneEdge** is able to use the network proximity estimation mechanism to perform latency-sensitive application placement. The placement done by **OneEdge** is cognizant of scarce Edge resources as well and only places the latency-sensitive components on the Edge.
- **OneEdge** effectively uses distributed monitoring mechanism to detect violations of response-time constraint due to client mobility in a timely manner. **OneEdge**'s root-cause analysis policy is able to identify the source of violation and trigger a migration to alleviate it.
- **OneEdge** uses the dynamic spatial context management mechanism to perform application placement ensuring perfect spatial alignment. In the event of workload surge at a particular application worker, the monitoring mechanism is able to detect the violation of response-time constraint and trigger workload partitioning.

6.7.1 Evaluation Scenario

Application Workload

We use two situation-awareness applications as the driving workload for evaluations.

- **Drone:** This application performs collaborative mapping of 3D space by a swarm of UAVs. The application contains logic for immediate-term control of the movement of each UAV (that is latency-sensitive), along with the stitching together of maps from individual UAVs (that is latency-tolerant). It is based on the work of Samira Haya et al. [55] and Alex Zihao Zhu et al. [56]. 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 three stages:

1. Feature tracking and detection from the IMU and cameras inputs.
2. Pose state estimation from the features extracted (update).
3. Updating the global map created by merging information from multiple drones in the swarm.

For our evaluations, we use a dataset generated using the ROS [57] framework for both the inputs of the camera and IMU [58]. To model the mobility of the drones (each drone operates independently), we use the Luxembourg City vehicle mobility dataset [59], associating individual car mobility with that of a drone.

- **View-Fuse.** This is the collaborative perception application for autonomous driving, based on Zijian Zhang et al. [60]. This application fuses the objects detected by multiple autonomous vehicles from their respective fields of view to create an expanded world sub-regional view, that 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 [61]. 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.

We run both the applications against the input dataset and profile the execution time of each component. Table 6.1 and Table 6.2 show the compute profile of different components the Drone and View-Fuse application respectively that we will be using for this evaluation study.

Edge Infrastructure considered

The infrastructure considered for this evaluation study is based on the metropolitan area of Luxembourg City. We assume 6 Edge site locations in the city, with site capacities derived from Alibaba Edge Node Service dataset. The locations of Edge sites is derived by k-means clustering on the cell tower locations in the city, and therefore, each cell tower is mapped to a unique Edge site. Each client is directly connected to the Edge site to which its currently serving cell tower is mapped to, that we refer to as the client's home site. The network RTT

Table 6.1: Drone application's compute profile.

Component	Max Clients	Mean Processing Latency	CPU Req.	Memory Req.	Latency Threshold
Global Map Update	8	50 ms	4	2 GB	-
Pose Estimation	8	35 ms	3	1.5 GB	50 ms
Feature Tracking & Detection	8	2 ms	1	512 MB	-
Client Component	-	1 ms	-	-	-

Table 6.2: MapFusion application's compute profile.

Component	Maximum Clients	Mean Processing Latency	CPU Req.	Memory Req.	Latency Threshold
Fusion	16	60 ms	4	2 GB	100 ms
Filter	1	5 ms	2	1 GB	-
Detection (on Client)	-	20 ms	-	-	-

between the client and the home Edge site is 5 ms. The RTT between two different Edge sites is 50 ms, while the RTT between an Edge site and the Cloud datacenter is 60 ms. The Cloud datacenter contains virtually infinite amount of resources.

6.7.2 Evaluation Platform

In this dissertation, we are concerned about the quality of control-plane decisions taken by **OneEdge** with the aid of the proposed mechanisms. In order to perform large-scale evaluations of the control-plane decisions, we rely on discrete-event simulation of the **OneEdge** platform, the clients and Edge infrastructure. The simulation has been created using SimPy [62]. All delays in the system, such as network traversal latency, computation latency, control-plane decision-making latency, etc. have been profiled from the implementation of **OneEdge** and plugged into the simulated environment so that it faithfully represents the real-world.

6.7.3 Evaluation Results

We evaluate the control-plane of **OneEdge** with a workload that contains a variable number of clients of the two applications.

Application Placement Policy Evaluation. We first compare the goodness of its latency-sensitive application placement approach against a greedy placement approach. The greedy baseline places all the application components – latency sensitive or not – on Edge resources, that would deplete them in the event of increasing client activity. We perform this experiment under stringent Edge capacity scenarios, with each Edge site having a capacity of 64, 128 and 256 cores respectively. For this evaluation, we restrict the client workload to include only clients of the Drone application, because it is the only application with a latency-tolerant component. The number of clients is varied from 16 to 512, and we monitor the observed response time at the Pose Estimation component (see Table 6.1). Fig. 6.5 shows the fraction of clients that face SLO violations due to the depletion of Edge

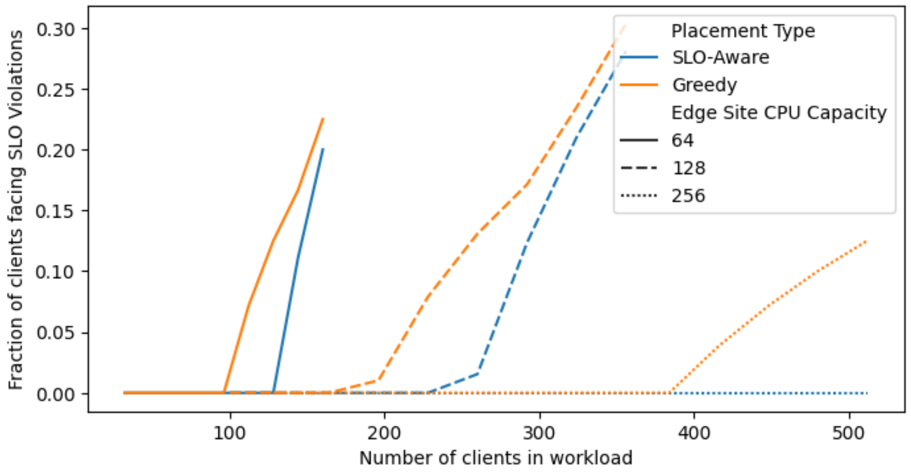


Figure 6.5: The fraction of total clients facing latency violations due to depletion of Edge resources for different placement policies and Edge site capacities.

resources for the two evaluated placement policies. For each Edge site CPU capacity, the Greedy placement policy results in client violation with fewer number of clients than the SLO-Aware placement policy. Scenarios with non-zero fraction of clients with SLO violations are those in which the capacity at the Edge sites has been depleted and subsequent application instances are deployed in the Cloud datacenter. Furthermore, intuitively, this depletion of Edge site capacity occurs with a higher number of clients as the Edge site capacity is increased. Therefore, we can prove evaluation hypothesis 1, that states that the placement policy of **OneEdge** is able to perform SLO-aware application placement, while also judiciously using scarce Edge resources only for latency-sensitive components.

Alleviating Client-mobility-driven Violations. We now show the behavior of observed response time from the Pose Estimation application component of the drone application in Fig. 6.6. The initial placement of the application components by **OneEdge**'s placement algorithm is such that the latency constraint of the Pose Estimation component is satisfied. However, over time as the drones move in the city, they enter the coverage area of a different Edge site than the one currently serving it. This results in an increase in the communication latency to the running application component instances, resulting in a violation of the latency constraint.

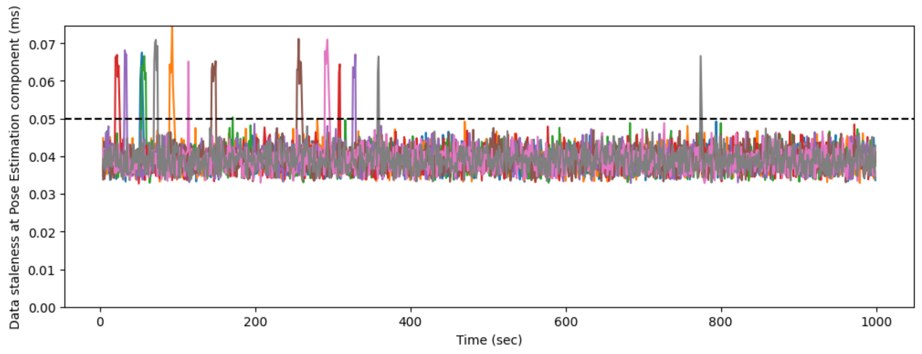


Figure 6.6: Variation of the observed data staleness at the Pose Estimation component of the Drone application over time. Only a few representative clients, that undergo mobility-driven violation have been shown for clarity. The latency threshold for the Pose Estimation component has been marked by the horizontal dashed line.

OneEdge's violation detection policy is able to utilize the end-to-end monitoring mechanism to detect that the Pose Estimation component's latency constraint has been violated for a given client. The root-cause analysis policy isolates the source of the violation to be the network latency between the Client Component and the Feature Tracking and Detection Component. It then triggers a client migration, that maps the violated client to an application pipeline instance that can meet the latency constraint.

Continuous Spatial Alignment. We now show how **OneEdge** is able to manage instances of spatially constrained applications while continuously providing near-perfect spatial alignment. Firstly, **OneEdge** performs application placement in a spatial-affinity-aware manner, mapping clients located in the same KD-Tree tile to the same application instance. This results in high spatial alignment values as shown in Fig. 6.7. Due to the natural mobility of vehicles, there are times when a large number of cars congregate in a small area, that results in more clients being served by the same Fusion component instance than expected. This causes an increase in the end-to-end processing latency at that component,

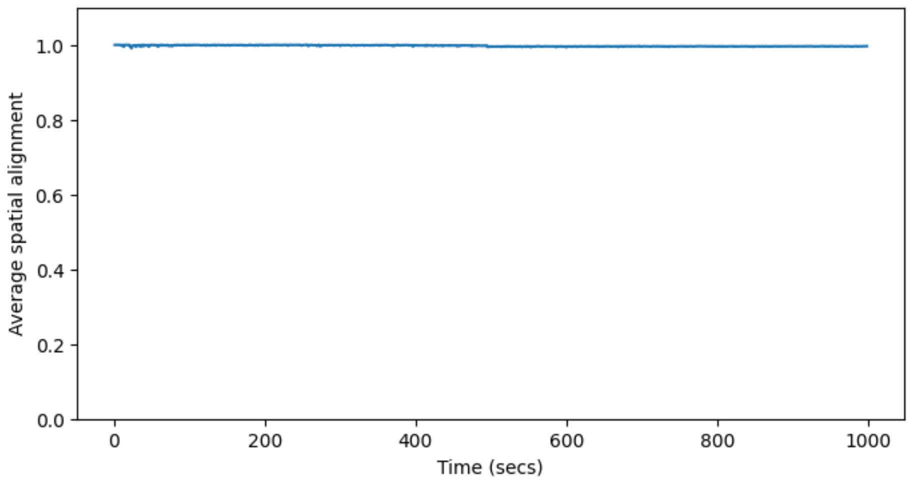


Figure 6.7: Spatial Alignment provided by **OneEdge**'s placement policy over time. The average spatial alignment is calculated every second by taking the average of the spatial alignment of each tile during that 1 second interval.

resulting in a violation, as shown in Fig. 6.8. Just as for the Drone application, the violation detection policy and the root-cause analysis policy utilize the end-to-end monitoring mechanism to determine that the root-cause of the violation is compute overload on the Fusion component. **OneEdge** triggers a partitioning of the set of clients served by the overloaded component in a way that spatial alignment is maintained. The tile mapped to the overloaded Fusion component instance is split and a new instance is created. This leads to the observed latency return to being under the threshold. All points of time when such a split is carried out have been marked by vertical dashed lines in Fig. 6.8.

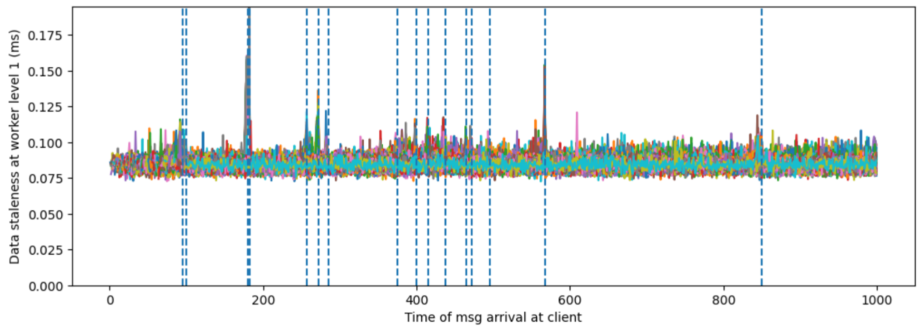


Figure 6.8: Response time from the Fusion component over time.

6.8 Related Work

There are a large number of research papers that propose latency-aware application placement policies on Edge infrastructure. Lera et al. [63] propose an availability-aware application policy and utilize network latency estimates between Edge sites. Naas et al. [64] propose a graph partitioning-based heuristic for data placement strategies to serve IoT applications. Deng et al.[65] propose a workload allocation policy that aims to attain a trade-off between power consumption and application delay on Edge infrastructure. Although these works show an improvement over the state-of-the-art in terms of achieving the applications' latency requirements, they do not cover how the network latency estimates were obtained, which are of central importance to their policies.

There have also been previous work on building full-blown application orchestrators for an Edge infrastructure. The Polaris Scheduler [66] has a control-plane design that is similar to **OneEdge**. It contains a central controller which maintains the infrastructure state as a graph data structure, using which it performs the placement of applications with complex end-to-end latency requirements. Application developers are allowed to specify arbitrary latency requirements between application components, which is a generalized version of the kind of constraints supported by **OneEdge**. The control-plane contains plugins that provide, among other things, network proximity information to the application placement policy. However, the work only qualitatively describes the information exposed by the plugins and does not discuss the interface between them and the control plane in detail. It also does not discuss the implementation of these plugins and their overhead on the system.

Rausch et al. [67] present a container scheduling approach for serverless Edge computing. The proposed system aims to schedule containers on Edge sites that are in proximity to the container registry as well as in proximity to the nodes that store data needed by these containers. This system has certain drawbacks in the context of serving situation-awareness applications. Firstly, the system does not take into account the latency of the data-plane. Proximity to container registry and data nodes only ensures that the container startup and application initialization are fast, but does not ensure that applications' latency requirements are satisfied. In addition, since the system is designed for short-lived serverless containers, it does not contain the relevant mechanisms for continuous management of deployed containers.

The Hetero-Edge system [68] is designed to perform orchestration of real-time vision applications on an Edge infrastructure. It targets infrastructure that can be heterogeneously endowed with computing resources ranging from CPUs to GPUs. Hetero-Edge uses the programming model of Apache Storm to break down an Edge application into multiple smaller tasks. It performs latency-aware scheduling of tasks on Edge nodes by offline profiling of the compute latency of tasks at different CPU utilizations and the bandwidth usage of each task. This step is similar to the application placement policy in OneEdge. The latency of a task on a node is the sum of processing latency and network transmission latency. Another control decision taken by Hetero-Edge is stream grouping, i.e., choosing which downstream application worker is to be chosen for the output stream. However, Hetero-Edge does not consider propagation latency, which is the most important difference between system entities in Edge infrastructure. They don't support client mobility and their evaluation scenario is not geo-distributed at all.

6.9 Conclusion

In summary, **OneEdge** is an application orchestator for situation-awareness applications on Edge infrastructure that is able to satisfy their response-time and spatial affinity requirements. It utilizes the network proximity estimation mechanism to perform latency-sensitive placement of application components on the infrastructure and the dynamic spatial context management mechanism for mapping clients to application instances while satisfying spatial affinity requirements. It leverages the end-to-end monitoring mechanism for detecting violations of response-time requirement, identifying the root-cause of the violation and triggering the right reconfiguration action. Through end-to-end evaluations of realistic application workload on a realistic infrastructure topology, we have shown that **OneEdge** is able to meet the requirements of situation-awareness applications.

CHAPTER 7
FOGSTORE: A GEO-DISTRIBUTED KEY-VALUE STORE GUARANTEEING LOW LATENCY FOR STRONGLY CONSISTENT ACCESS

Situation-awareness applications maintain state, that guides the action of the application in the future. Application state consists of recently generated events (such as a vehicle detection) and location-tagged data-items (such as the state of a traffic light). Hence, system support primitives for *saving and retrieving events* that constitute the application state are critical in a service platform meant for geo-distributed situation-awareness applications. Contemporary stream processing platforms like Foglets [18], Apache Flink [69] and Samza [70] provide primitives for accessing and modifying application state. Quite often, multiple application components on different Edge sites may want to share application state – e.g., situation-awareness applications may have multiple processes working on events pertaining to the same geographical area; this would require moving the state out of memory into an out-of-core external store [71] - a design choice that is also supported by Apache Flink. Keeping such application state information on Cloud-based data stores would defeat the purpose of placing the application components on geo-distributed Edge sites since saving/retrieving state is in the critical path of the application execution and should incur as little latency as possible. Hence, the application state needs to be stored in a geo-distributed manner as well [72], leveraging the same Edge sites as those used for placing the computational components.

Situation-awareness applications have stringent timing requirements on the sense-process-actuate control loop, which involves accessing application state in the critical path. To ensure that the application's control loop can operate at the desired speed, access to application state needs to be possible with low latency. Situation-awareness applications require strong consistency guarantees on the application state [73]. In fact, researchers at Google observe that coping with eventual consistency in the application layer requires significant development time and often leads to complicated and error-prone mechanisms [74]. Hence strong consistency should be provided by the datastore layer itself. Furthermore, the application state should remain available in spite of failures of data store nodes.

Cloud-based data stores such as Cassandra, DynamoDB, etc. have a performant data-plane which offers low latency of data access. They also offer strong consistency guarantees along with fault-tolerance. However, Edge infrastructure poses peculiar challenges which are not present in Cloud datacenters, for which these systems are designed. Firstly, the network topology of the Edge infrastructure is highly heterogeneous, and low latency data access is only possible if the data replicas are located close to the client. Ensuring strong query consistency demands that the data replicas on which query is executed are located in proximity to the client. However, Edge infrastructure is more susceptible to geographically correlated failure, which increases the probability of making multiple replicas of a given data-item unavailable. Hence, latency, consistency and failure tolerance are conflicting objectives at the Edge, and the control-planes of off-the-shelf data stores do not offer a reasonable tradeoff between them.

FogStore tackles the problem of providing both the conflicting but valid objectives of fault-tolerance and low-latency while satisfying consistency guarantees in geo-distributed key-value stores. The key insight about situation-awareness applications is the dependence of relevance of a certain data-item on the client's spatial context. For instance, in the smart city domain, information related to a particular city may be relevant to clients who are in close proximity to that city. Using this context-sensitive characteristic of applications, we design a replication strategy that guarantees strong consistency for spatially-relevant data items while also providing tolerance from geographically-correlated failures. The strategy is to place replicas close to the relevant clients (for low latency) and also in geographically distant locations (for fault-tolerance).

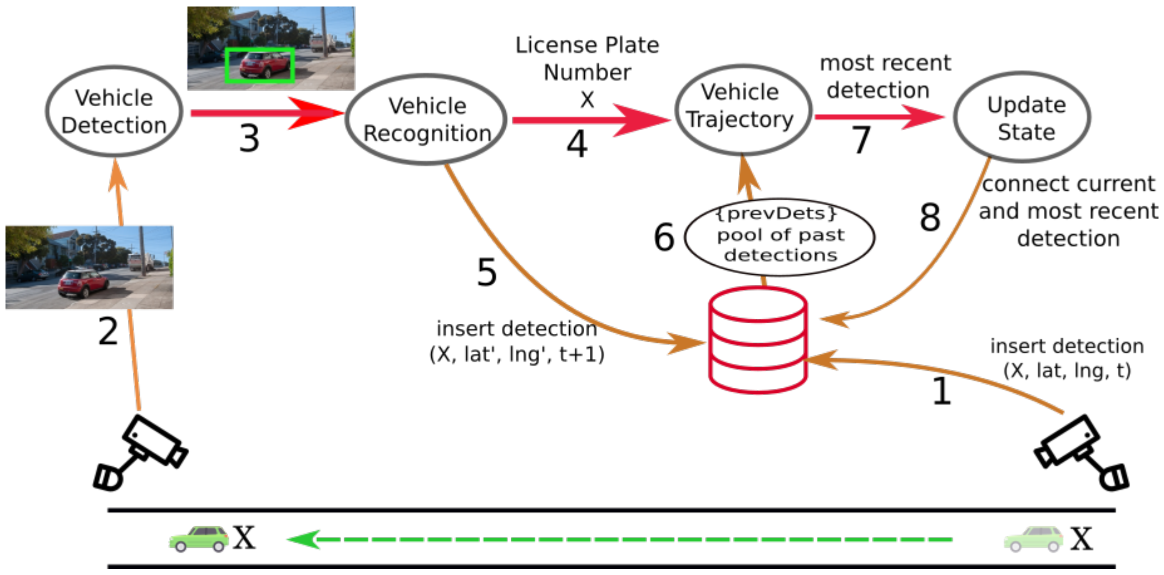
FogStore, which embodies the design principles for achieving fault-tolerance and low-latency for strongly consistent access to application state makes the following contributions:

- Develop a notion of *relevance* for situation-awareness applications, which is formalized as *Context-of-Interest* (CoI) - which determines the degree of consistency at which queried state should be reported;

- FogStore utilizes the Dynamic Spatial Context Management mechanism to perform location-aware replica placement to ensure low latency data access along with tolerance from geographically correlated failures.

- FogStore utilizes the Dynamic Spatial Context Management mechanism to perform quorum selection that guarantees strongly consistent operations for *relevant* data.

The roadmap of this chapter is as follows. Section 7.1 provides a background of the motivating application scenario and the basic concepts of a key-value store. It also discusses in detail the requirements of situation-awareness applications and the challenges of meeting



Vehicle Detection
Vehicle Recognition
License Plate Number X
Vehicle Trajectory
most recent detection
Update State
{prevDets} pool of past detections
connect current and most recent detection
insert detection (X, lat', lng', t+1)
insert detection (X, lat, lng, t)
1
2
3
4
5
6
7
8
X
X

Figure 7.1: Schematic of multi-camera tracking of suspicious vehicles. The numbered arrows suggest the temporal order of flow of information.

those requirements in an Edge setting. Section 7.2 presents the architecture of FogStore. Section 7.3 describes how FogStore leverages the Dynamic Spatial Context Management mechanism to perform replica placement to ensure low-latency data access along with tolerance from geographically correlated failures. Section 7.4 describes how FogStore uses the same mechanism to perform quorum selection to ensure high consistency data access for *relevant* data items. Section 7.5 presents results of experimental evaluation of FogStore and Chapter 9 concludes the chapter.

7.1 Background

7.1.1 Motivating Application Scenario

To highlight the necessity of a system like FogStore, we present a motivating use-case that poses strict latency and staleness requirements on the data-store. Consider a distributed camera network that may be deployed on urban roadways, feeding real-time video streams for multi-camera tracking of suspicious vehicles.

The application can be logically partitioned into components, as per the MobileFog programming model [75], each performing a specific function and having well-defined input-output characteristics. A schematic of the application is presented in Fig. 7.1. Upon detection of a vehicle in a video frame, the application extracts the identity of the vehicle by reading the license plate to get the unique identifier of the vehicle. It then inserts the detection of that vehicle along with location and time into the set of detections. The application also maintains the complete trajectory information of the vehicle, in that, for each detection of a vehicle, it saves the location and time when the car was detected before that. To achieve this, for each detection the application retrieves the set of previous detections that took place within 5 KM and 10 minutes from the current detection. It looks for the most recent detection from them and adds the identifier of that detection to the current detection's $prevDetection$ field.

Algorithm 16 Vehicle tracking algorithm

```
1: procedure onDetectVehicle(V, X, Y, T)
2:     $K \leftarrow$ INSERT INTO vehicle_detections $(V, X, Y, T)$
3:     $prevDets \leftarrow$ SELECT * FROM vehicle_detections
   WHERE $(x, y)$ within 5 km $\&t$ WITHIN 10 min ORDER BY t
4:     $prevDet \leftarrow k.V$ : $k$ has most recent time in $prevDets$
5:     UPDATE K in vehicle_detections SET K.prevDet = prevDet
```

It is evident from the schematic that access to application state lies in the critical path of application execution, hence making correct execution of the application contingent on fast access to state. For instance, if the insert of a vehicle detection is slow, the vehicle may be detected by an adjoining camera and not mark the former detection as the previous one - hence missing that detection from the trajectory. It is worth noting that the presented application has a dependence on contextually relevant events, specifically previous detections within a 5 km radius and at a maximum of 10 minutes before the current detection. Events that don't fall under this space-time filter may be past detections of the car but are typically not relevant for generating a fine-grained trajectory. Furthermore, retrieving the entire set of previous detections of the particular car and searching for the most recent detection could be slow.

7.1.2 Interface offered by a Key-Value Store

Data Model

The data-model of FogStore follows a key-value pair model, wherein keys are strings and values can be elementary data-types such as strings, integers or floating point numbers, or a dictionary of key-value pairs themselves. Following are the types of key-value pairs that data-items of typical applications contain.

- Each data item needs to possess spatial information, that is, location in terms of latitude-longitude. This field denotes the location of the event or entity that the data-item is about.

- Data-items, especially those that pertain to events that happened at a certain point of time, contain a timestamp field.
- Data-items are allowed to contain key-value pairs specific to the application's logic, such as vehicle identifier in the case of a multi-camera vehicle tracking application.
- It is up to the developer to specify which keys of a set of data-items constitute the primary key, which will be used to uniquely identify data-items for queries.

```
{
  "vehicle_id": "A54 3527",
  "location": {
    "latitude": "33.42553",
    "longitude": "-84.74456",
  },
  "timestamp": "1520123197"
}
```

Listing 7.1: A sample data-item that captures a spatio-temporal event in the vehicle-tracking application's detection set. The field storing Vehicle ID forms the key that will be used to retrieve detections of a particular vehicle.

Types of Query Supported

FogStore supports the following types of queries.

- **Insert/Delete data-items.** FogStore allows applications to insert data-items into the store by providing the key-value pairs that make up the data-item, including the primary key. For deleting a data-item the application can simply use the primary key.
- **Update data-items.** FogStore allows applications to update an existing data-item by providing the primary key of the data-item and the new set of key-value pairs. If the key exists the value will be updated, otherwise a new key-value pair will be added to the data-item.

- **Selecting data-items.** Applications can fetch data items whose key-value pairs match a certain predicate. The predicate can either be an exact match on one or more keys or an inequality-based condition.

Replication and Consistency of Data

Dynamo-style data stores have the notion of keyspaces, which denotes the namespace of keys. Key-value pairs belonging to the same application have keys that belong to the same keyspace. Dynamo-style data stores map data to storage nodes by computing a hash-function on the key. The output range of the hash function is divided among the storage nodes such that if a key's hash falls within the partition that is mapped to a particular storage node, then the key-value pair will be mapped to that node. Each storage node is assigned a *token* on the output range of the hash function. The storage node with the smallest token that is greater than the hash value of a key is assigned as the primary replica of the key-value pair (as shown in Fig. 7.2). Additional replicas are chosen based on the requirements of the application (replication factor, rack-awareness, etc.) and the primary replica.

Key-value stores built on the Dynamo-style model offer fine-grained tuning options for consistency per-operation. Fig. 7.2 shows how Cassandra (a popular Dynamo-style key-value store) distributes data among the cluster nodes. The data-item's key is hashed to determine the nodes responsible for storing it (replicas). The client making read/update requests can specify a consistency level for that specific operation, which determines the number of nodes that need to execute that operation before the client is acknowledged of its completion. For example, an update operation with consistency level of TWO would update the copy of the data-item on 2 of the replica nodes and acknowledges that the update was successful. The update is propagated to the rest of replicas in an eventual manner. The choice of this consistency level plays a crucial role in tuning the tradeoff between latency and consistency. Eventually consistent implementations use a consistency level of ONE while strong consistency implementations use consistency level of QUORUM. In a highly geo-distributed datastore, synchronizing with all replicas for each operation can be extremely slow, given that the replicas may be stored on nodes with a high network round-trip-time from the client or the coordinator node.

7.1.3 Control-plane Decisions involved in a Key-Value Store

FogStore is a geo-distributed key-value store with data stored on multiple Edge sites. FogStore needs to make the following control-plane decisions:

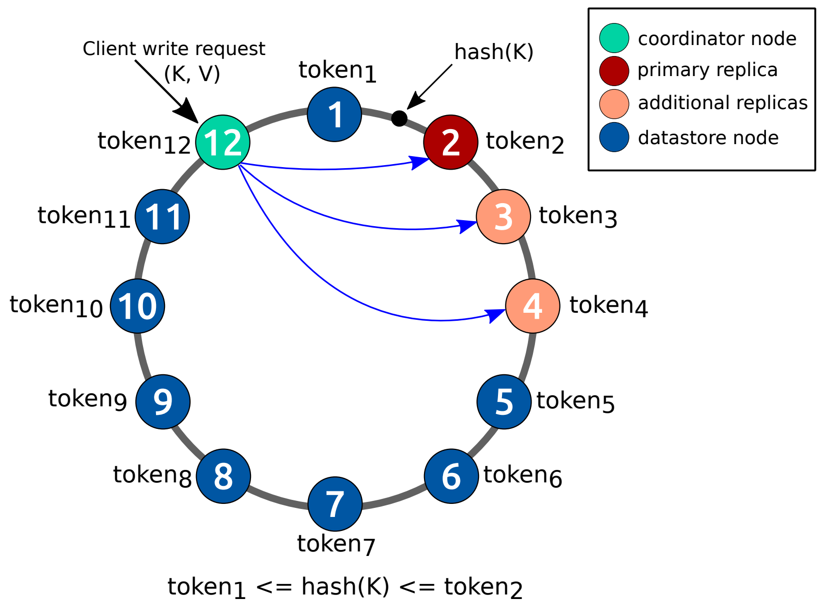


Figure 7.2: Demonstration of write-request path for a 12-node Cassandra cluster on keyspace with Replication Factor (RF) = 3

- **Replica Placement.** FogStore needs to determine the data store nodes that should host the replicas for a particular of data-item.

- **Quorum Sizing.** For each query, FogStore needs to decide how many of the existing replicas of the queried data-item should be contacted before returning the result to the client. For read queries, the query coordinator node reads the data-item from Q replicas of the data-item and returns the latest version to the client, where Q denotes the quorum size selected. Similarly in the case of an insert or update query, the query coordinator returns success to the client as soon as the query has been executed on Q replicas.

7.1.4 Application Requirements

- **Low-Latency of Data Access.** State access is in the critical-path of typical applications. Hence, low-latency data access would allow the application instances to maintain their response-time.

- **High Consistency.** For those data-items that have multiple readers and writers, applications require that the version of data that they read is the latest version. Reading a stale version of the data could result in errors in the application logic. Ensuring strong consistency in the application layer requires significant development time and often leads to complicated and error-prone mechanisms [**shute2013f1**]. Hence, the data store needs to be responsible for providing data access with strong consistency guarantee.

- **Failure Tolerance.** Applications assume that with sufficient number of replicas of each data-items, at least one copy of a data-item will always be available for accessing in the event of failures.

7.1.5 Challenges in achieving Application Requirements

Operating a data store over a geo-distributed Edge infrastructure has a number of unique challenges arising out of the nature of the infrastructure, and are discussed as follows:

- Network topology of the Edge infrastructure is heterogeneous, with Edge sites connected to the Internet through different peering points, thereby making the latency between Edge sites heterogeneous. Hence, the data-placement strategies used in cloud-based data stores that ensure uniform load-balancing across storage nodes would result in high data-access latencies at the Edge.

- High-consistency data read and writes require performing operations with multiple replicas of a data-item. In an Edge-based data store, where the inter-node latency distribution is heterogeneous, low-latency data access with high-consistency can only be achieved if all the replicas in the quorum are located close to the query coordinator.

- Edge infrastructure consists of sites that often share the same network and power lines. These sites are susceptible to geographically correlated failures. Hence, placing all replicas of a data-item on nodes in proximity of each other to ensure low-latency high-consistency access would result in making that data-item more vulnerable to such failures. In such an event, all replicas of that data-item would be lost.

- Data-items that compose the state of situation-awareness applications are generated due to activity in the physical environment. Spatial skews exist in the distribution of activity. For instance, within a city, there may be surveillance cameras deployed only in a certain portion of the city (e.g. Downtown), which would create a skew in terms of the number of events pertaining to those areas compared to rest of the city. Storing data-items on storage nodes that are located close to the location of the data-item itself would result in workload skews, that would impact query performance and increase the storage requirement on a subset of nodes. This problem is avoided in datacenter-based data stores by using consistent hashing [76] to distribute data-items across storage nodes, but that would significantly deteriorate query latency in a geo-distributed data store.

Hence, low-latency, high-consistency and tolerance from geographically-correlated failures are conflicting objectives in an Edge setting.

7.2 Architecture of FogStore

In this section, we first describe the edge-centric control-plane policies in FogStore at a high-level, that allow it to meet the application requirements. Then we discuss the functionalities of the system components in FogStore.

7.2.1 Edge-centric Control-plane Policies

FogStore's control-plane policies for replica placement and quorum sizing aim at simultaneously providing the application requirements of low-latency, high-consistency and tolerance from geographically-correlated failures, while dealing with the specific challenges of the Edge infrastructure. It does so by utilizing a peculiar characteristic in the data-access pattern of situation-awareness applications, wherein certain data-items are of higher relevance for the application logic if they are located in the *spatial context* of a given instance of the application. We call the spatial context of the application instance Context of Interest, which is explained further in the following.

Context of Interest

Situation-awareness applications have a strong notion of locality-awareness. For example, in the case of publish-subscribe systems, the publishers and subscribers are often located geographically close to each other, because the events they are concerned with pertain to the local geographical area. This property was exploited by Teranishi et al. [77] with the Locality-aware publish-subscribe system. In general, geo-distributed applications possess the notion of contextual relevance, in that a given data-item is more relevant in a certain context than another. The notion of context is application-specific. For instance, in the use-case mentioned in Section 7.1.1, a vehicle detection is relevant at a higher degree in a context around the event in space and time, that is, within 5 kms from the event's location and within 10 minutes of the event generation. This is because of the nature of the application - to build an accurate trajectory of a vehicle, the previous detection has to be in proximity to the current detection. Similarly, smart cars accessing the state of a traffic light (red/green), where the cars are located in proximity to the traffic light, would require consistent access to traffic light state to prevent collisions. Cars not in the traffic light's proximity would be able to live with data that may be stale, that is, eventual consistency may suffice for them. Often the data generated by Internet-of-Things applications are used for big-data analysis, which are offline batch processing tasks and don't require highly consistent data. In the context of key-value stores, the notion of relevance translates to consistency and staleness as follows : all relevant items must be available in a consistent manner with minimum staleness. In other words, for a data-item I all entities for whom I is contextually relevant must see updates to I in the same order (serializability) and with as low staleness as possible (real-timeliness).

We now formally define the above notion of contextual relevance. Fogstore allows the application developer to specify a generic (possibly conservative) *region of relevance* (also called Context of Interest) around each data item such that operations originating from that region are executed in a strongly consistent manner. We represent the region of relevance as a bounding box in geographical space. The size of the CoI is configurable and application-specific. Fogstore compares the location of a client to the location of the queried data-item and provides a strongly consistent result only when the client's location lies within the CoI of the data-item. Clients beyond the CoI are typically not using the data for critical operations and hence can tolerate inconsistent/stale data to the same extent as an eventually consistent database.

Two Replica Types offering Differential Consistency

We would like to revisit the two fundamental requirements of situation-awareness applications:

- Clients sharing context with the data-item require low-latency access to strongly consistent data.
- Placement of all replicas in proximity of the client may lead to complete data loss under geographically-correlated failures.

These requirements lead us the design decision of having two classes of replicas, which is also illustrated in Fig. 7.3.

1. **In-CoI replicas** : Placed on nodes located at low network delay from the clients, typically within/around the CoI of the data-item. Their maximum distance from the

data-item's location is determined by a parameter *InCoIDist*. This parameter determines the location of InCoI (strongly consistent) replicas and hence has implications on the latency of query operations. These replicas are kept consistent by enforcing that all read and write quorums include a majority of these replicas. These replicas are meant to provide both low-latency and strong-consistency to users that are contextually relevant to the queried data-item.

2. **Out-CoI replicas**: The purpose of these replicas is to provide tolerance from geographically correlated failures. They are placed on datastore nodes which are a minimum distance away from the data-item's location, a parameter called *OutCoIDist*. This parameter determines the location of OutCoI replicas, and hence affects the geographical separation between InCoI and OutCoI replicas - and thereby the fault-tolerance. These replicas are kept eventually consistent by propagating updates asynchronously, and hence are never included in the read/write quorum operations originating from within the CoI. These replicas could also serve as the source of data for big-data analysis of information generated at the edge, by using tools like Kafka Connect [78] that can use key-value stores as data sources for large-scale analytics.

It is worth noting that the concept of differential consistency for replicas based on their context is not a novel concept proposed in this dissertation. Apache Cassandra itself provides a consistency level called LOCAL_QUORUM [79] that makes sure that any operation selects a quorum that consists of the quorum-set of replicas on the local datacenter on which the request-handler (coordinator) node is located. This is done to avoid the high latency of inter-datacenter traversal. The updates are propagated to rest of the replicas (in other datacenters) in an eventual manner. However, this concept is easy to adopt in the context of cloud-based data-stores, which have a well-defined notion of datacenters such that the network latency across datacenters is at least an order of magnitude higher than intra-datacenter latency. Data stores deployed on densely geo-distributed Edge infrastructures cannot make use of such a concept, especially when the concept of contextual locality is based on node location and geographical distance rather than simpler properties like a datacenter identifier or IP address subnet.

Handling skews in event workload

FogStore's replica placement policy ensures that InCoI replicas are located close to the location of the data-item, while also ensuring that the workload across data-storage nodes

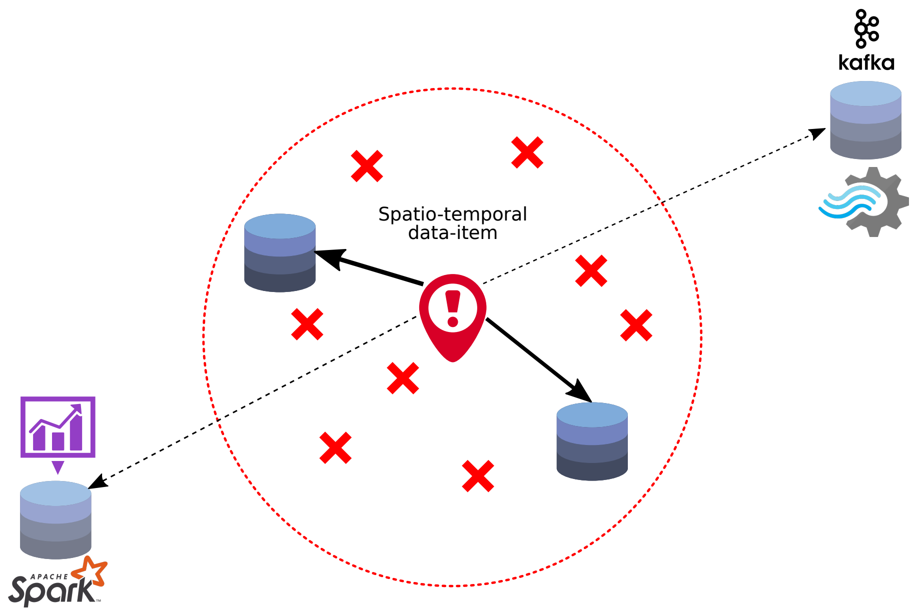


Figure 7.3: Illustration of the two types of replicas maintained by Fogstore along with the typical use-cases that both these types serve. The dotted circle around the spatio-temporal data-item denotes the Context-of-Interest for the data-item. We direct all reads/updates to that data-item originating within the CoI (represented by red crosses) to the InCoI replicas (which are shown inside the circle). The other (OutCoI) replicas are kept eventually consistent and serve the use-cases dealing with batch processing and monitoring.

is balanced. It does so by incorporating a hash-based replica placement approach akin to datacenter-based data stores into the location-based policy. More concretely, among the candidate nodes that fulfil the condition of InCoI replicas being close to the data-item's location, FogStore uses hashing to map the replica to tge candidate node. By doing so, FogStore not only achieves location-awareness but also ensures that the workload is evenly distributed among all candidate storage nodes. The notion of proximity is tunable, and can be set to match the degree of skew in workloads.

Hence the major issues that the implementation of Fogstore should resolve are :

- Placement of replicas based on the data-item's context, so as to have replicas both in proximity serving the queries from relevant clients within the CoI with low-latency and also at a significant geographical-separation from the CoI to provide tolerance from geo-correlated failures.

- Providing a transparent consistency interface to clients by determining the per-query consistency level based on contextual information of the client and queried item. This tuning of consistency-level is done by choosing the quorum of replicas in a way so as to deliver strongly consistent information to relevant clients and possibly stale information to clients outside the context-of-interest.

- Avoid the formation of hotspots in data-partitioning due to inherent skews in the input workload.

7.2.2 FogStore Client

The client of FogStore is an entity in the situation-awareness application pipeline that needs to read or update state. It could be an end-client, such as an application module running on an autonomous vehicle querying the status of a traffic light in vicinity, or a module running on an Edge site as described in Section 7.1.1. Since the client is a component of a situation-awareness application, it is associated with a certain spatial context. The client provides the current spatial context to FogStore along with the query request. FogStore system determines if the current spatial context of the client overlaps with the Context Of Interest of the data-item being queried to determine the appropriate consistency level for executing the query.

7.2.3 FogStore Servers

FogStore is composed of multiple geo-distributed servers which act as the interface between the clients and the stored data. These servers perform two main functionalities. Firstly, they act as Coordinators for query execution, wherein they perform the entire execution of the query on behalf of the client. The coordinator checks the location of the client and the CoI of the data-item being requested to determine whether the data-item is contextually relevant to the client. As discussed in Section 7.2.1, if the client is contextually relevant, the quorum set for this query is set to a majority of the InCoI replicas (strong consistency), otherwise the quorum is set to 1 (eventual consistency). In other words, the coordinator would return the query result to the client after a quorum of replicas have returned the result.

Second, FogStore servers act as Data Storage Nodes – whereby they host replicas of data-items and respond to queries from the coordinators. Each data storage node is responsible for a partition of the hash-space (as described in Section 7.1.2) based on its token. Since FogStore performs location-aware replica placement, the token of a data store node is derived from its location.

7.3 Use of Spatial Context Management mechanism for Replica Placement

In this section, we show how the Spatial Context Management mechanism is used for selecting replica placement candidates for a data-item. As described in Section 7.1.2, in Dynamo-style data stores, each data-item is hashed to generate a scalar value which is used to select the right node to host its replica. In FogStore's case, since replica selection needs to be based on location, the location field of data-items and storage nodes need to be converted into a scalar value to serve as the hash lookup key and node token respectively. The key idea in FogStore's use of the Spatial Context Management mechanism is to use the spatial partitioning created by the mechanism as a geo-index to encode location to a scalar value.

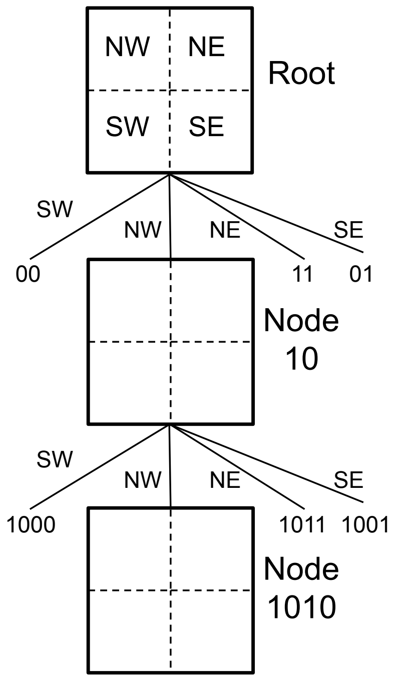
7.3.1 Using Spatial Partitioning for Spatial Encoding of Geographical Locations

We partition the entire geographical space containing the application workload and data store nodes using the Spatial Context Management mechanism. The partitioning is then used to geo-index data-items and storage nodes. Each vertex of the KD-Tree in the proposed spatial partitioning is assigned a unique ID based on the area that it covers. The root of the tree covers the entire geographical area in question, while the size of the smallest tile or leaf vertex is determined by the height of the KD-Tree. Fig. 7.4a illustrates such a KD-Tree of height 2. Each non-leaf node is partitioned along the two axes - latitude and longitude - and therefore has 4 children. These children are termed as $child_{sw}$, $child_{nw}$, $child_{ne}$, $child_{se}$ respectively. Fig. 7.4b illustrates how the ID of each child vertex can be derived from the ID of the parent vertex. We convert the ID of each node using base-32 encoding to a human-readable ID. The height of the KD-Tree is configurable and can be tuned based on application-specific requirement of low-latency and uniform load balancing. FogStore then encodes the location of a data-item or a storage node to the ID of the tile that the location maps to.

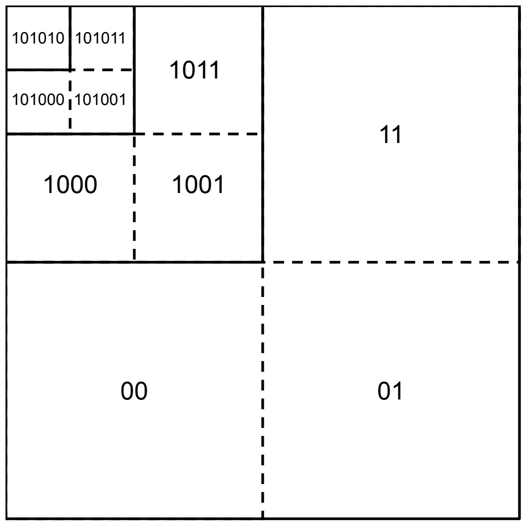
Each data-item has an additional field called $partition\_key$ which stores the ID of the leaf spatial partitioning tile that it belongs to. This ID forms the hash-value that is used to lookup the token ring for replicas.

7.3.2 Building the Token Ring

In order to perform location-aware data distribution we use the spatial encoding (as described in Section 7.3.1) of a data-item's location field to compose the partition-key. Data-



(a) KD-Tree of height 2 for partitioning geographical space. Each node of the tree is split into 4 parts - Southwest (SW), Northwest (NW), Northeast (NE) and Southeast (SE). This figure only shows the Northwest child of each vertex.



(b) Spatial partitioning obtained by using a KD-Tree of height 3. As in Fig. 7.4a, only the Northwest child of each vertex is expanded into its own children.

Figure 7.4: Illustration of using KD-Tree based spatial partitioning in FogStore.

store nodes are placed on the token-ring at a position equal to the spatial encoding of their location. In order to ensure that a data-item is placed on a node whose spatial-encoding is similar to the data-item's, it is important that tokens be ordered with respect to their spatial encodings for replica selection. Hence we don't use the popular consistent hashing algorithm for generating the token, but rather use the partition key directly as the token.

Notion of distance in spatial encoding

We define a notion of distance between two spatial encodings which is core to choosing the right nodes to place replicas on. Two spatial codes $d1$ and $d2$, have a distance of d if and only if $d1$ and $d2$ have the d^{th} bit as the maximally significant bit that differs in them. This is clarified in Fig. 7.5. This notion of distance can be applied to any spatial encoding technique used to calculate tokens. It preserves the closeness of locations, that is, if two locations are closeby in terms of the proposed encoding distance, they are also closeby in terms of geographical distance. The converse, however, is not true, that is locations that

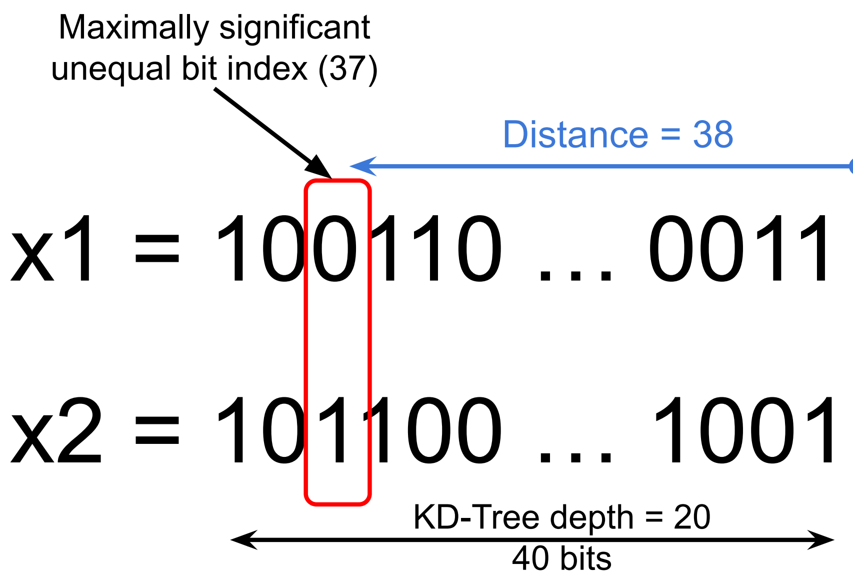


Figure 7.5: An illustration of distance between the IDs of two tiles in a KD-Tree of height 20.

are close in geographical distance may have spatial encodings that are far apart in terms of encoding distance. This property is evaluated in Section 7.5 when determining the location of OutCoI replicas.

7.3.3 Selecting Candidate Replica Nodes based on Data Item's Spatial Context

Selection of replicas for a given data-item is done based on the spatial encoding distance of a node's token and the data-item's token. The replica selection algorithm takes the following two parameters as inputs :

- *InCoIDist* : the maximum spatial encoding distance threshold for placing InCoI replicas.
- *OutCoIDist* : the minimum spatial encoding distance threshold for placing OutCoI replicas.

The first node token that the item's token gets mapped to in the token ring is used as the starting point for iteration to find the InCoI replicas. Each potential node that is within CoI's distance threshold is a suitable candidate. A similar search is performed for getting the list of OutCoI replicas, the difference being that the spatial encoding distance between item's token and node's token now needs to be greater than the CoI distance threshold.

7.3.4 Ensuring Even Load Distribution among Data Storage Nodes

The distribution of spatio-temporal event traffic is not expected to be uniform, with much more activity in densely populated regions and lesser activity in sparsely populated regions. Forming a node's token solely based on the spatial encoding would lead to partitions in the hash ring not being uniform in terms of the number of tokens contained in them. This can lead to uneven distribution (poor load balancing) of key-value pairs across data store nodes. Consistent hashing forms one extreme of data partitioning that achieves best load balancing, but does not take into account spatial locality of replicas, while just using spatial encoding forms the other extreme which guarantees spatial locality but not load balancing. Hence, the two objectives of proximal data placement and load balancing prompts us to come up with a hybrid data partitioning scheme. We construct the partition-key $part\_key$ of a data-item d as shown in Fig. 7.6.

$$\text{part\_key(d)} = \underbrace{\text{tile\_id(d.loc)}}_{g \text{ bits}} \cdot \underbrace{\text{mmh3(d.key)}}_{k \text{ bits}}$$

Figure 7.6: Illustration showing the inclusion of location-specific information in data-item's token to enforce proximal placement and the hash of key for even distribution.

7.3.5 Replica Placement Algorithm

The replica selection policy for InCoI replicas is presented in Algorithm 2. The first node token that the item's token gets mapped to in the token ring is used as the starting point for iteration to find the InCoI replicas. Each potential node that is within CoI's distance threshold and has a token with key portion lexicographically higher than item's token is a suitable candidate. Note that the comparison of token's key portion is solely for load balancing purposes. A fixed number of such replicas are selected. If the required number of replicas are not found after a complete traversal of the ring, the CoI's distance threshold is incremented by a small amount and the search is repeated. This increase in the threshold (which is specified by application developer) may harm the expected latency, but we choose to do so over declaring that no suitable replicas could be found.

Algorithm 17 Replica selection algorithm

```
1: procedure findPrimaryReplica($H, itemToken, it_S$)
2:     $it_P \leftarrow$ null
3:     for $it \in iterate(H, it_S)$ do
4:         if $itemToken \leq it \& it.key \geq itemToken.key$ then
5:             $it_P \leftarrow it$
6:             $break$
7:     if $it_P$ ==null then
8:         $it_P \leftarrow it_S$
       return $it_P$
9: procedure findReplicas($H, itemToken, it_P, inCoiDist, N$)
10:     $replicas \leftarrow \{\}$
11:     for $it \in iterate(H, it_S)$ do
12:         if $encodingDist(it.tile\_id, itemToken.tile\_id) \leq inCoiDist \quad \& \quad it.key \geq itemToken.key \quad \& \quad it$ not already chosen then
13:             $replicas = replicas \cup \{it\}$
14:         if $|replicas| == N$ then
15:             $break$
       return $replicas$
16: procedure findInCoIReplicas($H, itemToken, inCoiDist, nReplicas$)
17:     $it_S \leftarrow H.find(itemToken)$
18:     $it_P \leftarrow findPrimaryReplica(H, itemToken, it_S)$
19:     $nFound \leftarrow 0$
20:     $inCoiReplicas \leftarrow \{\}$
21:     while $|inCoiReplicas| < nReplicas$ do
22:         $R \leftarrow findReplicas(H, itemToken, it_P, inCoiDist, nReplicas - nFound)$
23:         $inCoiReplicas \leftarrow inCoiReplicas \cup R$
24:         $nFound \leftarrow nFound + |R|$
25:         $inCoiDist + +$
       return $inCoiReplicas$
```

7.3.6 Optimizations for efficient range queries

Typical Dynamo-style key-value stores designed for datacenters use consistent hashing (Murmur3 hash) on the partition-key to partition the key-value pairs across nodes. Since the Murmur3 hash function does not preserve the order between input values, they do not allow range queries on the partition key. However, Fogstore calculates partition key of data items as shown in Fig. 7.6, which preserves the order between partition-keys, thus making it possible to issue range queries on them. A typical range query has a structure as shown in Listing 7.2.

```
SELECT * FROM tracking_ks.detections
WHERE token(part_key)>=token(<min_part_key>) AND
token(part_key)<=token(<max_part_key>) AND
key='<object_key>'
```

Listing 7.2: Typical form of a range query that Fogstore handles. The minimum and maximum limits of partition key range queried for is calculated based on the bounding-box of locations that forms the range query. Note that we omit timestamp based filtering for simplicity, however that can be incorporated in the query provided a secondary index is built on the timestamp field.

For efficient execution of this query, a secondary index on the key column is created so that events within a particular partition can be queried in lesser time. Upon receiving such a query, the coordinator node splits the range into individual partitions and issues concurrent read requests to the replicas responsible for those partitions.

7.4 Use of Spatial Context Management mechanism for Consistency Tuning

Selection of replicas constituting a quorum based on the context of interest of the data item being queried is done based on the token of the coordinator node [1]. We use a parameter *CoISize*, which represents the size of the CoI in terms of spatial encoding distance. If the coordinator's token is less than *CoISize* units distant from the item's token, it lies within the CoI of the queried data item, and a quorum of InCoI replicas are selected to sychronously execute the operation. This ensures that all operations within the CoI will be highly consistent. Read operations on coordinators that are outside the CoI of queried data item are returned the version from whichever replica responds the fastest, thus providing eventual

[1]The underlying assumption is that client and coordinator node are in relative proximity

consistency. Write operations, on the other hand, need to update a quorum of the InCoI replicas before returning so that InCoI replicas don't enter an inconsistent state. Enforcing quorum for operations inside the CoI is fast as the quorum members are located in close network proximity from the client/coordinator node.

7.5 Evaluations

We implement the architecture present above by extending Apache Cassandra. We use the distributed network emulator MaxiNet [80] to create the infrastructure setup for evaluation experiments. MaxiNet is an extension of the popular network emulator MiniNet, and allows the user to package an application as a Docker container and deploy it on a set of nodes. Latencies between the nodes are calculated using the geographical distance between the nodes based on the online tool WAN Latency Estimator[2] and emulated using Linux Traffic Control *tc*.

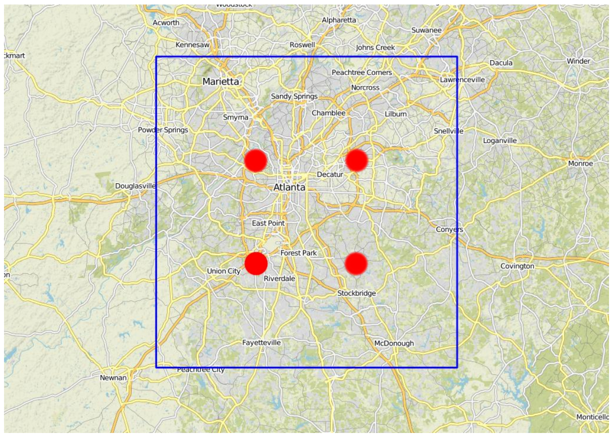


Figure 7.7: Map showing the locations of Edge sites in Atlanta region and the bounding box from where events are generated. The sites shown in the figure are the ones that store the InCoI (consistent) replicas in Fogstore. The evaluation also consists of nodes located at remote locations : Houston, San Francisco, Chicago and Seattle.

7.5.1 Comparison of FogStore against typical replication policies

The first step towards proving the efficacy of the context-aware policies of Fogstore is to evaluate its performance against typical replication policies - quorum-based (strict) and eventual[3]. For this experiment, we build a representative infrastructure topology of data-store nodes - 4 inside Atlanta region and 4 more as remote datacenters. Yahoo Cloud

[2]http://wintelguy.com/wanlat.html

[3]The strict and eventual systems use Cassandra's SimpleStrategy policy for replication.

Serving Benchmark (YCSB) is used to generate spatio-temporal workloads of applications running in the Atlanta area, by having 4 client nodes running YCSB with 4 threads each. Each YCSB client is collocated with one of the datastore nodes in Atlanta area and mimics a situation-awareness application component making queries to the spatio-temporal state. To cover a wide variety of workloads, we experiments with workloads that have varying read-to-update ratios (20%, 50% and 80% reads) and varying distribution of selecting keys for operations (hotspot, latest and Zipfian). We consider mutable data-items to measure the behaviour of evaluated systems on consistency. We set the replication factor of eventually consistent and quorum-based store to 3. For the replication policy of Fogstore, we set number of InCoI replicas to 2 and OutCoI replicas to 1. Also, the read and write quorum for queries by clients inside the CoI are set to 2 and 1 respectively.

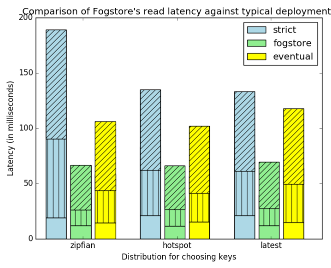
To be able to perform these experiments and collect the required metrics, we had to modify the core workload executor of YCSB. Here we describe the major modifications to YCSB here :

- We associate each version of an entity with a given key with a timestamp, which corresponds to the wall-clock time when the YCSB client starts updating or inserting that version. When updating the entry in the table, we also add the timestamp associated with that version to that entry. We also record the time when an update/insert finishes. Since the experiment is done as an emulation on a single machine, the clocks of all datastore nodes and YCSB clients (Docker containers) use the time of the underlying physical machine.

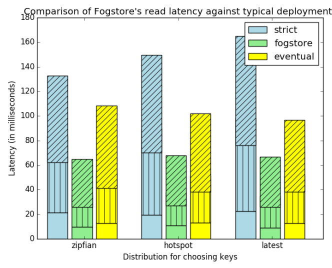
- We associate each key generated by YCSB to a unique geolocation, since the spatial attributes of a data-item is central to the consistency model of Fogstore. The challenge here is to create a deterministic mapping between the key domain and the geolocation domain - so that this mapping stays the same across all the threads on all YCSB clients. Each key selected by the request distribution of YCSB is based on an integer value, which we convert to a sequence of bits. This bit sequence is then decoded (exactly same as spatial decoding using KD-Tree) to generate the latitude and longitude of the request.

- We needed to modify Python's driver for Cassandra to allow queries to have CoI-aware consistency level.

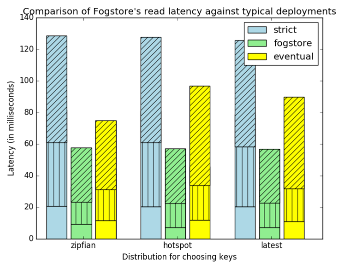
- We use the latency aware load balancing policy to avoid choosing a coordinator that is too far away in terms of network latency - effectively defeating the purpose of



(a) 20% reads

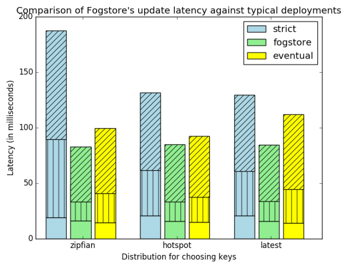


(b) 50% reads

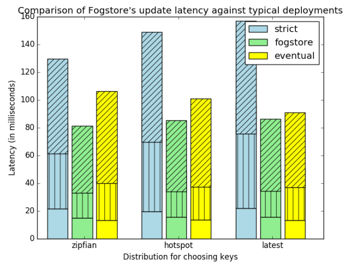


(c) 80% reads

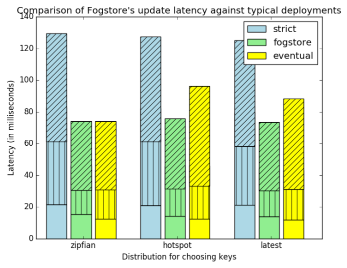
Figure 7.8: Read latencies for varying distribution of reads. The solid, vertically dashed and obliquely dashed portions of the bar denote 50^{th}, 95^{th} and 99^{th} percentiles respectively.



(a) 20% reads



(b) 50% reads



(c) 80% reads

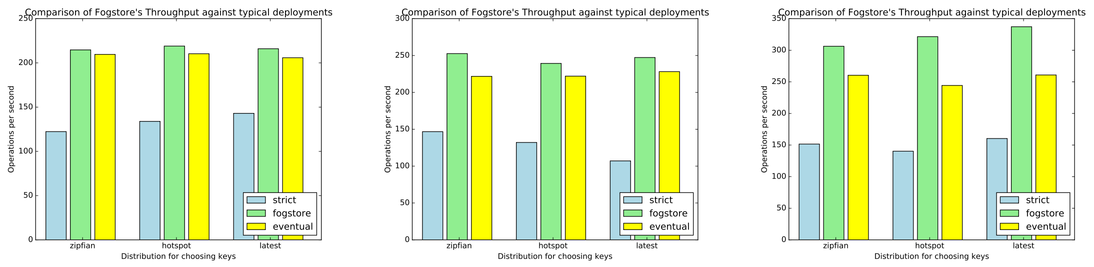
Figure 7.9: Update latencies for varying distribution of reads. The remaining operations are updates. The solid, vertically dashed and obliquely dashed portions of the bar denote 50^{th}, 95^{th} and 99^{th} percentiles respectively.

Fogstore.

The metrics we are interested in are :

- Latency of read/update operations
- Throughput of read/update operations
- Client-centric degree of consistency

Applications that require strong consistency for data-accesses would use the aforementioned quorum-based (strict) datastore. The client's expectation from a datastore guaranteeing strong consistency is that a read always returns the most recent version whose update was successful. We analyze the trace of YCSB clients and for every read compare the version returned to the version that was written by the most recent successful update, and call it a violation if these versions don't match. We analyze the YCSB client trace against a quorum-based store and, following the expectation, don't find any violations as the read



(a) 20% reads (b) 50% reads (c) 80% reads

Figure 7.10: Throughput (ops/sec) for varying distribution of reads

Table 7.1: Percentage of reads returning a version that was not most recently written. The percentage of inconsistent reads has been reported for workloads with varying proportion of read requests and key-selection distribution.

	20% reads	50% reads	80% reads
Latest	0.17	0.6	0.11
Hotspot	0.06	0.06	0.02
Zipfian	0.03	0.02	0.01

and write quorums overlap. This means that the application does not have to implement special logic for handling inconsistent/stale data reads.

This reduction in programming effort due to strong consistency guarantees from the datastore comes at a price on performance, as now each operation has to wait to complete on a quorum of nodes, which may have high network latency between them. This hypothesis is validated by the variation of read and update latencies shown in Figures Figure 7.8 and Figure 7.9 and the aggregate operation throughput shown in Figure Figure 7.10. The applications in this paper's context are heavily dependent on the low-latency execution of read and update operations, and hence guaranteeing performance of paramount importance.

For the sake of performance, the applications at hand would use an eventually consistent datastore, and wait for each operation to complete on only one replica before acknowledging the user. The experimental results show that the eventually consistent datastore is able to outperform the quorum-based store significantly in terms of operation latencies and throughput (Figures Figure 7.8, Figure 7.9 and Figure 7.10). However, since clients may initiate reads before previous updates to those data-items have been propagated to all replicas, there are mismatches between the version returned by read operations and the most recently written version. Table 7.1 shows the percentage of reads that undergo such consistency violations. Note that due to the limitations of the emulation platform, we only have 16 client threads in these experiments, and increasing the concurrency would lead to more reads that violate strong consistency. Hence the improvement in performance comes at the cost of programming effort to handle inconsistent reads from the datastore.

To counter the performance limitations of quorum-based and consistency violations of an eventually consistent datastore, we run the same client workloads against Fogstore, which is fundamentally Cassandra extended with the context-aware policies for replication and quorum selection. Since the read and write quorums are limited to replicas placed close to the data-item and overlap, there are no consistency violations i.e., it offers strong consistency. On the performance side, we observe that read and update latencies are even better than that of the eventually consistent store. We reason that this is because of the location-agnostic replica placement in plain Cassandra, where all the 3 replicas of a data-item may be located on remote nodes, leading to high operation latency even with eventual consistency. Fogstore is, therefore, able to achieve a better throughput than the eventual store. In fact, since the read quorum is set to 1 and write quorum to 2, the throughput for a read-heavy workload beats the eventual datastore by a higher margin as the number of replicas to block for is just one.

Hence, based on these results, the context-aware optimizations of Fogstore allow it to obtain the performance of an eventually-consistent store and, at the same time, provide consistency guarantees similar to a quorum-based database. This enables the design of systems that are dependent both on high throughput and low programming effort of dealing with inconsistent operation results.

7.5.2 Performance of range queries

Applications processing spatio-temporal data have a high performance dependence on efficiency of range-queries. Fogstore performs data-distribution based on location, which would lead range queries to span across a number of datastore nodes. In the following set of experiments we analyze the performance of Fogstore for delivering results of range queries and the impact of range filter size. We compare the performance of range queries on Fogstore against a baseline eventually consistent database setup - which uses the data-item's *type* to partition items across datastore nodes. Since events are not mutable, we are not dependent on consistency guarantees.

For this paper, we assume that range queries request events of a certain type T in a circular geographical area of radius R km centered at location L and can be expressed as the tuple (L, R, T). To support efficient range queries, we built a wrapper that transforms the tuple (L, R, T) into a number of contiguous subset of tile IDs, such that they cover the area covered by the range (similar to [**spatialcassandra**]). For each of these subsets we trigger a *child* concurrent sub-query to Fogstore that returns events with a location falling under the subset of tile IDs. The range query is said to be completed when all the *children* sub-queries are complete. For this set of experiments, we build a datastore cluster same as the previous experiment, with 4 nodes around Atlanta and 4 at remote locations (see Figure Figure 7.7). A KD-Tree of depth 10 is chosen for spatial partitioning. We load the database with 4000 events of 10 different types from the area marked by the blue outline. For each range query (L, R, T), L is sampled from the bounding box in Figure Figure 7.7 while T is sampled from the set of 10 event types. The value of R is varied and the impact of its variation on range query completion time is reported in Figure Figure 7.11.

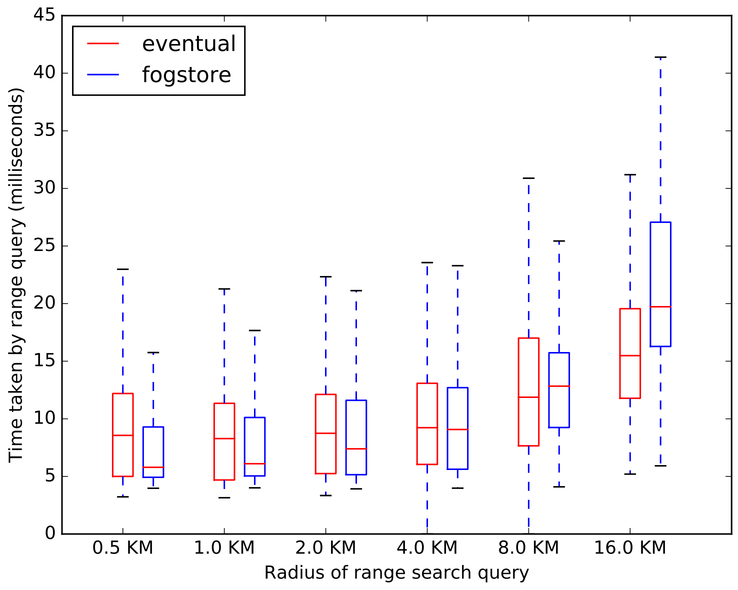


Figure 7.11: Comparison of latency of range queries with varying range-radius for Fogstore's data partitioning and eventually consistent Cassandra deployment. All statistics are aggregated over 5000 range queries.

As is evident from Figure 7.11, the location-based distribution of data done by Fogstore is able to achieve comparable performance to the baseline eventually consistent datastore with the increase in radius of range queries. Increasing the range-query radius increases the number of replicas (datastore nodes) that would store the events queried for, which increases the overhead on Fogstore's coordinator node to split the original sub-query further down into read requests that are sent to individual replicas. This factor, however, does not impact the baseline datastore, since the events are partitioned based on the item type field, which is fixed for a particular range-query, meaning each replica assigned to that item type would have all the data-items of that type. Furthermore, an increase in range-query radius also leads to increase in the number of events returned, which also impacts query-completion time, both for Fogstore and the baseline.

7.5.3 Load balancing across data-store nodes

One of the supposed limitations of partitioning data-items based on spatial encoding using Cassandra's hash ring mechanism is that skews in application workload can lead to some datastore nodes storing significantly more data-items than others. This is because of the fact that we want to preserve spatial proximity of replica placement.

In order to perform large-scale tests of load balancing, we design a simulation environment which mimics the replica placement approach of Fogstore [4]. We focus on the region around Atlanta, as shown in Figure Figure 7.7, and place 64 datastore nodes within that region in a uniformly-spaced manner. To simulate spatial skew in data access pattern, we generate 80% of the data-items from a small area (0.0625 times the full region) around Downtown while the rest of the region generates 20% of data-items. For every data-item we set the number of InCoI replicas to 2, and configure the CoI distance so as to have all the 64 fog nodes candidates for hosting the InCoI replicas. Furthermore, we also envision having multiple datastore nodes deployed at a particular resource location, akin to a mini-datacenter. The data-partitioning policy should be able to utilize all the available capacity, both at the same location as well as across multiple locations. The metric of interest is the number of data-items that each node is assigned. A data partitioning policy that does not take load-balancing into account would lead to nodes close to the hotspot region storing much more data-items than those away from it, resulting in a high variance in the aforementioned metric. We report the measured metric in Figure Figure 7.12.

As described in Section 7.3.2, the depth of the KD-Tree used for spatial partitioning also lends itself to determining the spatial encoding of a data-node's location. The KD-Tree height is, hence, crucial for determining the degree of spatial load balancing. A smaller value of height leads to a large number of nodes having the same location-specific part of the token, leading to more widespread load balancing. A larger height leads to higher spatial proximity and, hence, less widespread load balancing. We are able to see the above effect of the KD-Tree height on the degree of load balancing (as show in Figure Figure 7.12). Interestingly, a KD-Tree based encoding with height of 12 and above encodes the location

[4]The emulation platform used in previous experiments is not large enough to emulate a very large number of nodes

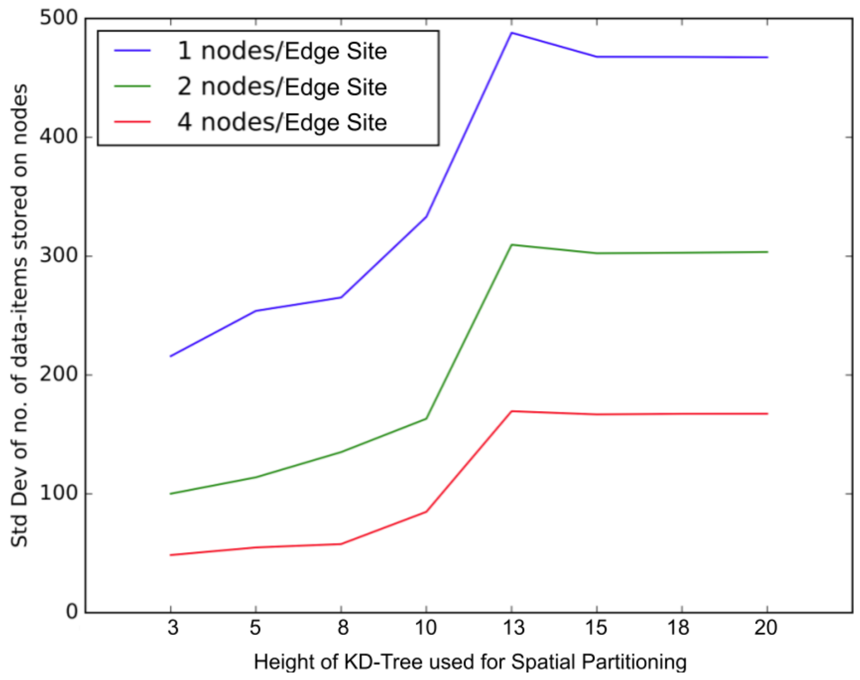


Figure 7.12: Standard deviation of the number of data-items stored by each datastore node (lower number denotes better load balancing). We vary the height of the KD-Tree used for spatial partitioning and encoding as well as the number of datastore nodes on each resource location. The numbers are averaged over 10 simulation runs.

of all datastore nodes to be distinct, thus leading to no changes in load balancing metric with trees of greater height.

Furthermore, an increase in the number of datastore nodes on each resource location also improves the load balancing metric, as the data-partitioning policy is able to seamlessly distribute data-items across them as well. Hence with a proper KD-Tree height token formation and enough resources near regions with higher activity, we can obtain both spatial proximity of replicas and good load balancing.

7.5.4 Evaluation of fault-tolerance

One of the important properties of Fogstore's data distribution policy is that it takes tolerance to geographically correlated failures into account. Contemporary cloud-based databases provide such fault-tolerance by distributing replicas of a data-item across multiple datacenters, so that they are at a sufficiently high distance from each other to not be affected by correlated failures like earthquakes or massive power outages. Fogstore, however due to lack of physical constructs like datacenters and regions, performs wide-area geo-distribution using the location of data-items and data-store nodes, by guaranteeing a certain minimum distance in their spatial encodings. However, since spatial encodings tend to translate higher (two) dimensional attributes into one-dimension, high distance in terms of encoding of two locations does not consistently translate to high geographical (actual) distances between the two locations

In the following set of experiments we examine the distribution that the geographical distance between Out-of-CoI replica nodes have from the data-item. Since the In-CoI replicas are kept close to the data-item's location, placing the Out-of-CoI replicas significantly far away from the data-item's location implies geographical separation between the In-CoI and Out-CoI replicas - thus achieving the aforementioned tolerance to correlated failures. We choose a few candidate large-scale resource topologies that are likely of being representative of how such geo-distributed infrastructures could be realized, which are described below :

1. Capitals : Each capital of mainland USA's states is considered a central location of resource capacity, with a fixed number of fog nodes scattered around a certain radius (50 km) from the capital city's centre.

2. AT&T : Locations of core routers of AT&T's backbone network inside the USA are considered central locations of resource capacity, with a fixed number of fog nodes scattered around a certain radius (50 km) from the peering points location.

Accesses are generated from within a radius of 100 km from each central resource capacity location (an example of which is shown in Figure Figure 7.13). We note the

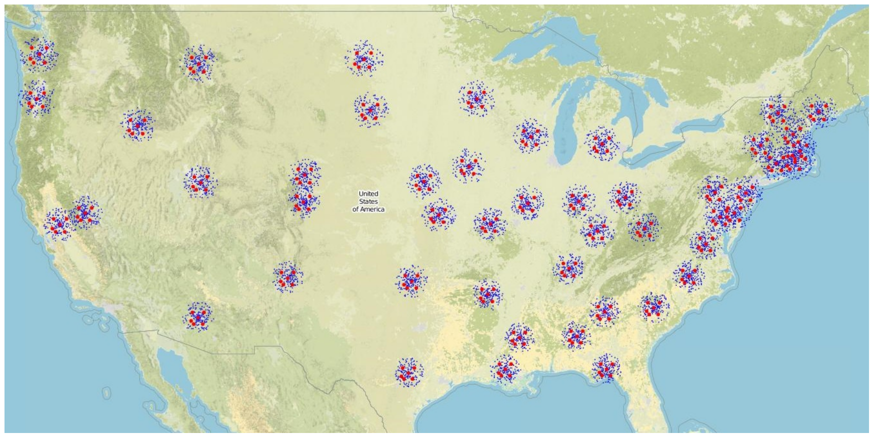


Figure 7.13: Geographical distribution of data-store resources (red dots) and accesses (blue dots) for a reference topology with mainland USA state capitals as resource capacity locations.

location where the Out-CoI replicas are placed and the distance of that node from the location of the data-item for the aforementioned reference topologies, as shown in Figure Figure 7.14. For both the candidate topologies, a maximum encoding distance threshold of 34 can allow a median separation of atleast 2500 KM, which is a significantly large distance for geographically-correlated failures, like natural disasters or massive power outages, to affect.

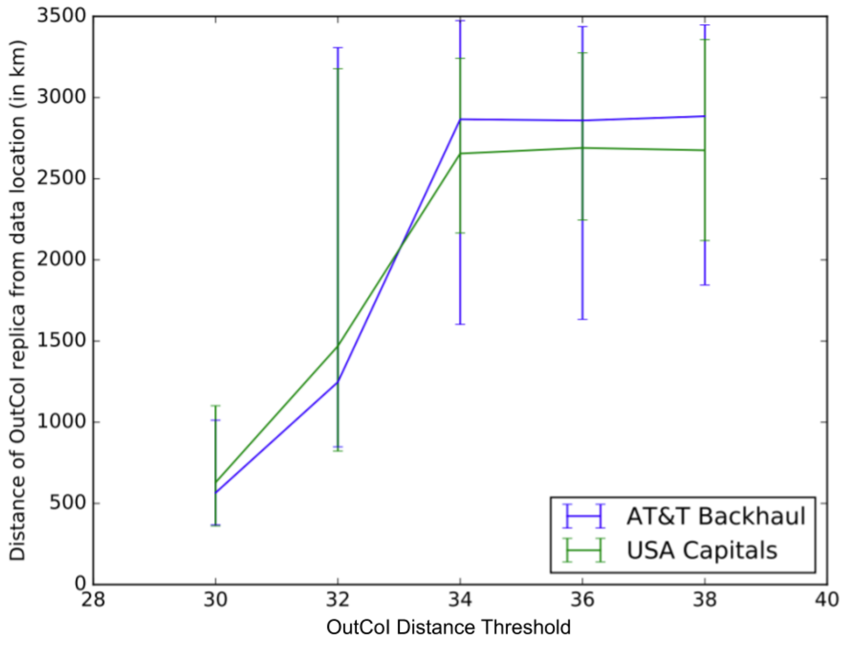


Figure 7.14: Distribution of distance between OutCoI replicas and data-items' locations for various reference topologies. The height of KD-Tree used for generating the node tokens has been taken as 20.

7.6 Related Work

ElfStore[81] is a resilient data storage service that is designed for a federated infrastructure consisting of both Edge and Cloud resources. It offers content-based discovery of data along with high reliability and transparent access irrespective of where the data is stored. The data placement policy of ElfStore ensures that placement is done within two hops from the client for fast retrieval. To provide high reliability of data storage, it monitors the uptime of each storage node and performs replication on a select subset of nodes such that a minimum reliability threshold can be provided. However, the system does not deal with mutable data and therefore has no notion of consistency tuning as FogStore. In addition, all replicas are placed in proximity of the client, therefore not offering any resilience from geographically correlated failures.

Cathode [82] presents a consistency-aware data placement algorithm for Edge-based data stores. It performs replica placement based on the consistency requirements of the application, the cost of synchronization between replicas and the storage capacity constraints on the data nodes. The system is designed for smart-city applications that have a strong geographical locality in data-access patterns. It uses pairwise network latency estimates between client and data store nodes to compute the cost of consistent read/write operations for a given data placement, while relying on a decentralized approach for computing the best placement for each data-item. The paper, however, does not cover the overhead of collecting the information of network latency at the different decentralized nodes for decision-making. The network-aware data-placement approach can be a meaningful addition to the system architecture of FogStore.

Portkey [83] presents an adaptive key-value placement approach over dynamic Edge networks. It monitors the latency between clients and datastore servers using efficient network tomography. An alternate placement for a client's data items is calculated only when the client's latency distribution to proximal datastore servers changes significantly. In case the placement needs to be updated in the event of client mobility, the data-items are migrated to a better data storage server. Although the proposed system does not support replication, PortKey's control algorithm is extensible to allow adding logic to perform placement of multiple replicas for avoiding geographically correlated failures.

EdgeKV [84] is a decentralized, scalable, and consistent storage system for the Edge. It supports two levels of data locality with different latency guarantees based on application requirements. The separation of local and global data allows deploying EdgeKV in different use cases. It is built for a layered infrastructure, with clusters of closeby Edge storage nodes connected to each other via gateways. Each cluster operates as a replicated state machine with distributed consensus for data consistency. The Gateways are part of a Distributed Hash Table (DHT) which uses consistent hashing for data lookups. Each data item is either Local or Global. In case it is Local, it is replicated in the local cluster, otherwise it is replicated in the Global DHT. The selection of Local or Global is coarse-grained, which makes the tradeoff between consistency and latency coarse-grained as well.

7.7 Conclusion

FogStore is a key-value store built for highly geo-distributed infrastructures with edge-centric context-aware replica placement and quorum selection policies – for both of which FogStore leverages the Dynamic Spatial Context Management mechanism. FogStore's replica placement is done taking the low-latency requirement of data-access into account, while also making the placement tolerant to geographically correlated failures. It, creates two types of replicas, ones which are located in proximity to the clients for low latency and ones in remote location for fault-tolerance. The quorum selection policy leverages the fact that the context of clients determine the degree of consistency expected when querying the state of a situation-awareness application. Hence queries from clients in the vicinity of the queried item, which require consistent data access, have a majority of the proximal replicas in quorum, while those from remote clients are offered eventual consistency. FogStore uses the Dynamic Spatial Context Management mechanism to perform both replica placement and quorum selection. We show the performance of the proposed policies by implementing them on Apache Cassandra and using YCSB to stress-test the system. Evaluations show that the proposed policies are able to achieve a throughput and latency comparable to eventually consistent systems, while still guaranteeing serializability guarantees on relevant data-items to clients.

CHAPTER 8
DISCUSSION

8.1 Realizing Proposed Mechanisms in Contemporary Edge Computing Offerings

In this chapter we discuss how the proposed mechanisms can be incorporated into the Edge platform offerings at present, and how they interact with the different stakeholders in the Edge Computing ecosystem.

8.1.1 Stakeholders in Edge Computing

The following stakeholders exist in contemporary Edge Computing offerings:

Infrastructure Provider. The most significant factor that differentiates Edge computing from the existing Cloud computing paradigm is the presence of a large number of geo-distributed infrastructure sites, which can be leveraged to offer compute, storage and networking services. Unlike building datacenters in a remote location, setting up Edge infrastructure capabilities is more complicated. This is because, firstly, multiple sites of real-estate have to be purchased in an urban area and power and network connectivity have to be ensured, all of which are expensive and cumbersome to manage. Secondly, since users of Edge applications would inevitably connect to the applications running on these sites through a telecommunication network, the Edge sites should be peered efficiently with the telecom network to avoid high access latencies, despite geographical proximity. Hence, there is a high bar for entry for newcomers in the Edge computing arena.

Fortunately, there are existing organizations who already manage such a geo-distributed infrastructure for their own purposes - telecommunication providers, such as AT&T and Verizon. They operate a large number of geo-distributed central offices which are stocked with network functions (NFV) or custom hardware that enable cellular connectivity. Recently, they have been opening up their infrastructure to be used for hosting Edge computing platforms. For instance, Verizon has partnered with Amazon Web Services to offer AWS Wavelength, which is an offering that provides AWS compute and storage services from Edge infrastructures located in several metropolitan areas in the USA [85]. Similarly, AT&T has partnered with Microsoft to provide Edge computing services [86]. Since these Edge sites are within the network of telecommunication providers, application clients would not have to pay extra latency overhead to communicate with them.

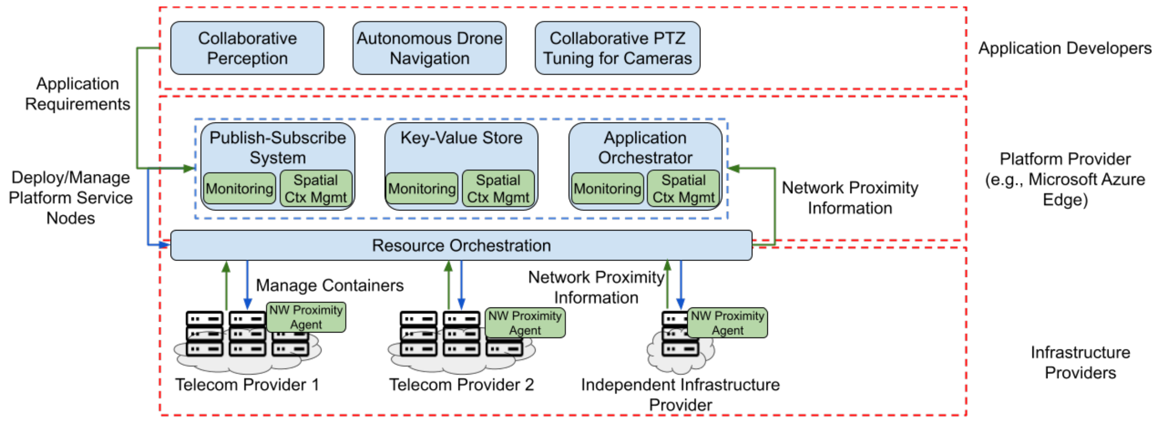


Figure 8.1: Schematic of the various stakeholders in an Edge computing environment and their interactions with one another.

There is a different flavor of infrastructure providers entering the Edge ecosystem. Examples of such infrastructure providers are Vapor IO [12] and Edge Micro [87]. These organizations build up facilities to serve as Edge sites and ensure efficient peering to network providers serving users closeby. Because of efficient peering, they are able to offer low-latency access to clients that are connect to them through other network service providers.

Platform Providers. These are organizations that utilize the infrastructure owned by Infrastructure Providers to run their application platform/middleware. Examples of platform providers are Amazon Web Services and Microsoft Azure, who already have experience in providing such middleware in the Cloud computing space. The middleware provided by them consists of several platform services that the applications can utilize, such as execution runtimes for applications, key-value stores for data storage and message queues for communication. Usually the same platform services that are offered by the provider in the cloud are made available at the Edge. Examples of such offerings are AWS Wavelength, Microsoft Azure Edge Zones, and Azure IoT Edge.

Application Developer. The application developer utilizes the platform services offered by the platform provider to implement the application logic.

8.1.2 Dependencies of Proposed Mechanisms on Stakeholders

We now discuss the kinds of dependencies that the proposed mechanisms have on the stakeholders of the Edge computing ecosystem. We will break down each mechanism into its constituent components and discuss which stakeholder manages that component.

Dynamic Spatial Context Management Mechanism

The Dynamic Spatial Context Management mechanism consists of two main components – (1) the spatial partitioning metadata managed by the control policy; and (2) the client library that maintains a cache of the current spatial tile and reports client location to the spatial partitioning metadata component. The spatial partitioning metadata has to be maintained within the control plane of the platform service that uses the mechanism. The client library of the mechanism needs to be incorporated into the client library of the platform service.

Network Proximity Estimation

The Network Proximity Estimation mechanism consists of three main components – (1) Edge Gateways; (2) Network Coordinate Agents on clients; and (3) Network Coordinate Agents on Edge sites. The Edge Gateways will have to be set up and maintained by the infrastructure providers as it is dependent on the network connectivity of clients and independent of the deployment of platform services. Network coordinate agents on the Edge sites would be managed by the infrastructure provider or the platform provider. In case they are managed by the platform provider, they would be able to have agents belonging to different infrastructure providers to be part of the same network coordinate cluster. Network coordinate agents on the clients could be a part of the client library of the platform provider itself, e.g., the client SDK of Azure IoT Edge. It could also be a part of the client library of the platform service.

End-to-End Monitoring

All components of the end-to-end monitoring mechanism would be deployed on the platform provider, possibly with one instance of the monitoring mechanism for each platform service.

8.2 Edge-Centric Architecture of Control-Plane

The mechanisms proposed in this dissertation aim at aiding the control-plane policies in their decision-making so that the data-plane of the platform services can be optimized for supporting situation-awareness applications. However, a move from the Cloud to the Edge not only affects the data-plane, but also the control-plane. In cloud-based implementations of platform services, the control-plane and data-plane are both resident within the same datacenter, and the latency between them is negligible. However in the Edge setting, these two components are separated potentially by a Wide Area Network. The high network latency between the control and data plane leads to the responsiveness of the platform service to deteriorate.

While this is not the focus of this dissertation, we make two specific optimizations in two of the platform services presented here to improve the responsiveness of the control-plane.

8.2.1 Increase Decision-making Autonomy at the Edge

OneEdge considers two classes of applications - *coordinated* and *standalone*. Coordinated applications are those that require multi-client collaboration, and hence scheduling decisions for them are taken in a centralized controller. On the other hand, each instance of a standalone application serves a unique client. Therefore, application placement for a standalone application client can be done independent of other clients. **OneEdge** assigns the responsibility of standalone application placement to the Site Agents themselves, which receive deployment requests directly from the clients.

Through experimental evaluations on real-world workloads in [88], we have shown that adding autonomy for resource scheduling improves the response time of the control plane. However, this autonomy results in the emergence of multiple writers to the aggregate infrastructure state maintained by the control plane, which needs to be consistent. One technique that allows high control-plane throughput in the presence of concurrent autonomous updates to the aggregate state by Site Agents is to maintain an eventually consistent view of the aggregate state at the central controller and use it to optimistically make scheduling decisions. These decisions then need to be verified against the remaining resource capacity at the Edge sites during Transaction Executor's operation. If the target Edge sites do not have enough spare capacity to carry out the scheduling decision, the decision is rolled back, aggregate state updated and the scheduling is redone. We have shown that the probability of conflicts between the central controller's and site agents' updates is low, which allows such a design to achieve high throughput and avoid overhead of rollback and rescheduling [88].

8.2.2 Improve Coordination between Control-Plane and Data-Plane

One of the important objectives of platform services serving situation-awareness applications is their ability to respond to performance violations by performing reconfigurations with agility. Such reconfigurations typically require multiple rounds of communication with the central control-plane and the data-plane components involved. Due to the high WAN latency between control and data-plane components, this process is slowed down, leading to the clients continuing to face performance violations for an extended period of time. **ePulsar** tackles this problem by making the communication between control and data-plane asynchronous, so that the multiple rounds of communication can proceed in parallel, thereby shortening the time required for performing the reconfiguration [89].

CHAPTER 9
CONCLUSION AND FUTURE DIRECTIONS

This dissertation proposed three mechanisms that aid the control plane policies of edge-centric platform services to satisfy the requirements of situation-awareness applications. This chapter first presents a summary of the main contributions of this dissertation and then presents future directions for research in this domain.

9.1 Conclusion

Situation-awareness applications consist of a sense-process-actuate control loop, which senses data from the environment, processes it and performs actions based on the extracted insights. These applications require low response time from the backend to ensure that the performed actions are in real-time with respect to changes in the surrounding environment. Furthermore, they also exhibit spatial affinity, wherein multiple nearby clients need to be served by the same application instance. These requirements are not fulfilled by the contemporary offerings of cloud-based platform services. Hence, new mechanism are needed to facilitate edge-centric control policy decisions in these platform services.

This dissertation introduces the proposed mechanisms in Chapter 3, wherein the necessity of each proposed mechanism are highlighted by quantitatively analyzing the performance of the state of the art control policies. It then formalizes the interface provided by each mechanism to the control policies of platform services, and demonstrates the utility of each mechanism in the context of a platform service's control policy. Chapter 4 then performs a design-space exploration of each mechanism. It presents candidate design choices, explains the metrics of interest and compares the candidate design choices against each other.

The dissertation then goes on to demonstrate the utility of the proposed mechanism in the context of three platform services in Chapters 5 through 7. Chapter 5 presents **ePulsar** , which is an edge-centric publish-subscribe system that offers end-to-end message delivery latency guarantees to applications. It utilizes the network proximity estimation mechanism for selecting the right broker for hosting a pub-sub topic. The mechanism is used for estimating the message delivery latency for a topic if a particular candidate broker is chosen for hosting the topic. It also utilizes the distributed end-to-end monitoring mechanism to monitor the observed latency by clients and trigger a topic migration to another broker if the latency requirement is violated.

Chapter 6 presents an application orchestrator system for situation-awareness applications. It supports applications that have response time requirements as well as spatial affinity requirements. It utilizes the network proximity estimation mechanism to perform latency-sensitive placement of application components on the infrastructure and the dynamic spatial context management mechanism for mapping clients to application instances while satisfying spatial affinity requirements. It leverages the end-to-end monitoring mechanism for detecting violations of response-time requirement, identifying the root-cause of the violation and triggering the right reconfiguration action. Through end-to-end evaluations of realistic application workload on a realistic infrastructure topology, we have shown that **OneEdge** is able to meet the requirements of situation-awareness applications.

Chapter 7 presents FogStore, which is a key-value store built for highly geo-distributed infrastructures with edge-centric context-aware replica placement and quorum selection policies – for both of which FogStore leverages the Dynamic Spatial Context Management mechanism. FogStore's replica placement is done taking the low-latency requirement of data-access into account, while also making the placement tolerant to geographically correlated failures. It, creates two types of replicas, ones which are located in proximity to the clients for low latency and ones in remote location for fault-tolerance. The quorum selection policy leverages the fact that the context of clients determines the degree of consistency expected when querying the state of a situation-awareness application. Hence queries from clients in the vicinity of the queried item, which require consistent data access, have a majority of the proximal replicas in quorum, while those from remote clients are offered eventual consistency. FogStore uses the Dynamic Spatial Context Management mechanism to perform both replica placement and quorum selection. We show the performance of the proposed policies by implementing them on Apache Cassandra and using YCSB to stress-test the system. Evaluations show that the proposed policies are able to achieve a throughput and latency comparable to eventually consistent systems, while still guaranteeing serializability guarantees on relevant data-items to clients.

9.2 Future Directions

We now discuss directions for future work that follow from the contributions of this dissertation.

9.2.1 Federation of Control Plane

This dissertation highlights the efficacy of the proposed mechanisms in the context of platform services that have a centralized control plane architecture, wherein policy decisions are made by a centralized entity. Note that by a centralized entity, we do not refer to a monolith. The centralized control plane could consist of multiple components, with replication to ensure fault tolerance. In this context, a centralized architecture refers to the fact that the control plane components are located in a single facility which is separated from all the data plane components by the wide area network. Such a design makes the coordination between control and data plane inefficient due to repeated WAN traversals to accomplish tasks such as application instance deployment, data migration, etc.

In a federated control plane architecture, there would be multiple controllers for a given platform service. Each controller would manage a subset of Edge sites, which could overlap with the set of sites managed by another controller. This overlap of sites is essential to ensure that in the event of the failure of a controller, the sites served by the failing controller can be managed by another one. When reserving Edge resource capacity for data or compute placement, the controller making this decision would need to synchronize with the other controllers managing the affected Edge sites to ensure that their infrastructure state is consistently updated. A distributed synchronization protocol such as two-phase commit can be used for this purpose.

A federated architecture allows the control plane to be located close to the Edge sites that host the data plane components, while still allowing the control policy decisions to be made against global infrastructure state. In our previous work [88], we have shown how introducing decentralization in the control plane results in better response times. This federation, however, comes at a cost in terms of how and when to handover monitoring data from one controller instance to another. These challenges can be addressed by once again leveraging the fact that the goal of a federated design is to have the multiple controller instances close to the Edge site they are managing. Hence, a shared data store for monitoring data could be utilized in which case explicit migration of data between controller instances can be avoided.

9.2.2 Extending to Other Platform Services

A widely used platform service in the Edge computing space is Functions as a Service (FaaS). FaaS is a useful paradigm for application development at the Edge because it quickly scales in and out according to ingress workload, and hence suits the scarce nature of Edge resources. Contemporary FaaS platforms such as Knative [90], Apache OpenFaaS [91] and OpenWhisk [92] consist of a centralized gateway, which receives ingress requests and distributes them to one or more workers of the requested function. Since they have been designed for datacenters, the communication latency between clients and the ingress gateway, and that between the gateway and worker nodes is predictably low. This assumption does not hold true in an Edge computing setting.

Therefore, to operate a geo-distributed FaaS platform, three main changes to the typical architecture are needed. Firstly, instead of a single ingress gateway, there need to be multiple geo-distributed gateways, with at least one gateway per Edge site hosting workers. This is to ensure that function invocation requests do not suffer from unnecessary network traversal overhead. Secondly, the dispatch of requests from the ingress gateway to workers should be done to ensure that the sum of communication and execution latencies does not exceed the application's end-to-end processing latency constraint. The estimation of communication latency between gateway and worker node can use the network proximity estimation mechanism proposed in this dissertation, while the current execution latency profile can be inferred using the end-to-end monitoring mechanism. Finally, worker nodes on Edge sites can get overloaded due to unpredictable workload surges. In such a scenario, the overloaded worker node would offload ingress request to other worker nodes which might have spare processing capacity. The choice of the node to offload a request is again dependent on the communication latency between the offloader and offloadee, such that the end-to-end latency constraint is not violated. For the estimation of communication latency, the network proximity estimation mechanism can be used.

9.2.3 Incorporating Radio Network Information into Control Plane Policy Decisions

This dissertation assumes that the access network link via a cellular tower's radio interface is reliable and offers stable network latency and bandwidth. However that is a simplifying assumption that does not hold in real-world scenarios. The performance of the access link varies due to interference with other user devices, user mobility or due to effects from the weather. Drops in channel quality causes packet loss, which affects both latency and throughput. The Mobile Edge Computing (MEC) standard by the European Telecommunications Standards Institute (ETSI) includes Radio Network Information Service (RNIS) [93]. The goal of the service is to allow authorized MEC application instances to consume RAN level information, such as UE channel quality indications and location updates, which they can utilize to offer enhanced services and optimize performance. Earlier this information was only available to the telecommunication network operator, who used it for allocating wireless spectrum resources to users. However, with the advent of MEC and the concomitant co-location of the control planes of the network and application stack, this information can also be shared with the application control plane stack.

The information provided by the RNIS can be used in a number of ways to offer reliable and stable performance to applications. For instance, Tan et al. [94] utilize the network throughput estimates from the RNIS to inform the cache update strategy for a Video-on-Delivery caching service running on the Edge. The key idea is to cache video segments of particular bitrates only, i.e., those bitrates that can be delivered to the user with good performance given the current network conditions. In a similar vein, Li et al. [95] propose a video delivery scheme for user devices that are connected to multiple access networks. The control plane of the proposed system periodically analyzes the status of the multiple access networks and decides which one to use for the delivery of a certain video segment so as to ensure optimal client experience. The control plane obtains network status using RNIS.

9.2.4 Monitoring Bandwidth as a Metric of Interest

This dissertation uses latency as the foremost performance metric for situation-awareness applications. However, for typical Edge computing applications, network bandwidth is also an important performance metric that should be taken into account when making control policy decisions. Several previous work propose bandwidth-aware control plane policies for managing applications that require high network bandwidth. Dedas [96] is an online deadline-aware task dispatching and scheduling algorithm that takes into account the network bandwidth and propagation latency between Edge sites. The objective of Dedas is to control the latency of task execution such that the deadline of the task can be met. It selects Edge sites for dispatching tasks such that both the transmission latency and propagation latency are low. Lavea [97] is a latency-aware video analytics platform which is designed for operation on Edge infrastructure. Lavea offloads video analytics tasks between clients and Edge nodes and also facilitates coordination between multiple Edge nodes. Based on the chosen offloading strategy, it allocates bandwidth among clients to ensure that the deadlines of tasks are met. The system consists of a monitoring service that periodically measures the latency and available bandwidth of the wireless link (4G LTE/5G/WiFi) which is used to make task offloading decisions. However, bandwidth monitoring is complicated as it involves sending probe packets through potentially already congested network links, which can severely affect application performance. The spatial and temporal variations in bandwidth can also be predicted in advance by using techniques such as those proposed in Foresight [98].

9.2.5 Proactive Violation Detection Policies

The control policies discussed in this dissertation for detecting violation of application requirements and root-cause analysis act in reaction to a violation. Therefore, clients will always go through a transient phase of requirement violation before the platform service can detect and alleviate it. However, proactive policies can be integrated into the control plane to proactively identify situations in which a violation is expected to occur in the near-future and trigger a re-configuration action. Such proactivity in the control plane would significantly minimize or even eliminate violations. The processing latency on application components can be analyzed to identify short-term trends which can help trigger a partitioning of client workload in advance. Systems like Foresight [98] can be used to predict when the network quality is going to deteriorate in the future and trigger a migration.

REFERENCES

[1] U. Ramachandran, H. Gupta, A. Hall, E. Saurez, and Z. Xu, “A case for elevating the edge to be a peer of the cloud,” *GetMobile: Mobile Computing and Communications*, vol. 24, no. 3, pp. 14–19, 2021.

[2] E. J. Saurez Apuy, “Control plane for situation-awareness applications on geo-distributed resources,” Ph.D. dissertation, Georgia Institute of Technology, 2022.

[3] H. Liu, P. Ren, S. Jain, M. Murad, M. Gruteser, and F. Bai, “Fusioneye: Perception sharing for connected vehicles and its bandwidth-accuracy trade-offs,” in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, IEEE, 2019, pp. 1–9.

[4] Q. Chen, X. Ma, S. Tang, J. Guo, Q. Yang, and S. Fu, “F-cooper: Feature based co-operative perception for autonomous vehicle edge computing system using 3d point clouds,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 88–100.

[5] H. Qiu, F. Ahmad, F. Bai, M. Gruteser, and R. Govindan, “Avr: Augmented vehicular reality,” in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, pp. 81–95.

[6] M. Boehme, M. Stang, F. Muetsch, and E. Sax, “Talkycars: A distributed software platform for cooperative perception,” in *2020 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2020, pp. 701–707.

[7] H. Huang, A. V. Savkin, and C. Huang, “Decentralized autonomous navigation of a uav network for road traffic monitoring,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 57, no. 4, pp. 2558–2564, 2021.

[8] J. Scherer, S. Yahyanejad, S. Hayat, E. Yanmaz, T. Andre, A. Khan, V. Vukadinovic, C. Bettstetter, H. Hellwagner, and B. Rinner, “An autonomous multi-uav system for search and rescue,” in *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, 2015, pp. 33–38.

[9] X. Meng, W. Wang, and B. Leong, “Skystitch: A cooperative multi-uav-based real-time video surveillance system with stitching,” in *Proceedings of the 23rd ACM international conference on Multimedia*, 2015, pp. 261–270.

[10] Z. Xu, H. S. Shah, and U. Ramachandran, “Coral-pie: A geo-distributed edge-compute solution for space-time vehicle tracking,” in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 400–414.

[11] T. Matsuyama and N. Ukita, “Real-time multitarget tracking by a cooperative distributed vision system,” *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1136–1150, 2002.

[12] V. IO. (Oct. 2021). “Kinetic grid, edge colocation and interconnection — vapor io.”

[13] 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*, 2021, pp. 37–53.

[14] CellMapper. (Aug. 2020). “Cellmapper.”

[15] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan, “Towards scalable edge-native applications,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 152–165.

[16] *IoT Edge cloud intelligence : Microsoft azure*.

[17] T. Lähderanta, T. Leppänen, L. Ruha, L. Lovén, E. Harjula, M. Ylianttila, J. Riekki, and M. J. Sillanpää, “Edge computing server placement with capacitated location allocation,” *Journal of Parallel and Distributed Computing*, vol. 153, pp. 130–149, 2021.

[18] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, “Incremental deployment and migration of geo-distributed situation awareness applications in the fog,” in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, 2016, pp. 258–269.

[19] S. Sarkar and S. Misra, “Theoretical modelling of fog computing: A green computing paradigm to support iot applications,” *Iet Networks*, vol. 5, no. 2, pp. 23–29, 2016.

[20] G. Amarasinghe, M. D. De Assuncao, A. Harwood, and S. Karunasekera, “A data stream processing optimisation framework for edge computing applications,” in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, IEEE, 2018, pp. 91–98.

[21] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, “Ifogstor: An iot data placement strategy for fog infrastructure,” in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, IEEE, 2017, pp. 97–104.

[22] G. Liu, J. Wang, Y. Tian, Z. Yang, and Z. Wu, “Mobility-aware dynamic service placement for edge computing,” *EAI Endorsed Transactions on Internet of Things*, vol. 5, no. 19, 2019.

[23] G. Ananthanarayanan, P. Bahl, P. Bodık, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[24] A. Das, S. Patterson, and M. Wittie, "Edgebench: Benchmarking edge computing platforms," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 175–180.

[25] S. Khare, H. Sun, K. Zhang, J. Gascon-Samson, A. Gokhale, X. Koutsoukos, and H. Abdelaziz, "Scalable edge computing for low latency data dissemination in topic-based publish/subscribe," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2018, pp. 214–227.

[26] A. Qureshi, "Power-demand routing in massive geo-distributed systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2010.

[27] J. Vilaça, J. Paulo, and R. Vilaça, "Geolocate: A geolocation-aware scheduling system for edge computing,"

[28] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.

[29] *Monasca - openstack*, https://wiki.openstack.org/wiki/Monasca, Accessed: 2022-05-09.

[30] Á. Brandón, M. S. Pérez, J. Montes, and A. Sanchez, "Fmone: A flexible monitoring solution at the edge," *Wireless Communications and Mobile Computing*, vol. 2018, 2018.

[31] T. Gonçalves, "Dynamic reconfiguration of the data aggregation topology at the edge," 2021.

[32] T. Rausch, S. Nastic, and S. Dustdar, "Emma: Distributed qos-aware mqtt middleware for edge computing applications," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2018, pp. 191–197.

[33] C. E. B. Bezerra, J. L. D. Comba, and C. F. R. Geyer, "A fine granularity load balancing technique for mmog servers using a kd-tree to partition the space," in *2009 VIII Brazilian Symposium on Games and Digital Entertainment*, IEEE, 2009, pp. 17–26.

[34] B. Donnet, B. Gueye, and M. A. Kaafar, "A Survey on Network Coordinates Systems, Design, and Security," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, pp. 488–503, 2010.

[35] S. Lee, Z.-L. Zhang, S. Sahu, and D. Saha, "On suitability of euclidean embedding for host-based network coordinate systems," *IEEE/ACM Transactions on Networking*, vol. 18, no. 1, pp. 27–40, 2010.

[36] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A Decentralized Network Coordinate System," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 15–26, 2004.

[37] H. Zheng, E. K. Lua, M. Pias, and T. G. Griffin, "Internet routing policies and round-trip-times," in *International Workshop on Passive and Active Network Measurement*, Springer, 2005, pp. 236–250.

[38] J. Ledlie, P. Gardner, and M. Seltzer, "Network coordinates in the wild," in *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, ser. NSDI'07, Cambridge, MA: USENIX Association, 2007, pp. 299–311.

[39] S.-Q. Lee and J.-u. Kim, "Local breakout of mobile access network traffic by mobile edge computing," in *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, IEEE, 2016, pp. 741–743.

[40] K.-J. Hsu, J. Choncholas, K. Bhardwaj, and A. Gavrilovska, "Dns does not suffice for mec-cdn," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020, pp. 212–218.

[41] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[42] C. Cañas, K. Zhang, B. Kemme, J. Kienzle, and H.-A. Jacobsen, "Publish/Subscribe Network Designs for Multiplayer Games," in *Proceedings of the 15th International Middleware Conference*, Bordeaux, France: ACM, 2014, pp. 241–252, ISBN: 9781450327855.

[43] S. Baidya, Z. Shaikh, and M. Levorato, "FlyNetSim: An Open Source Synchronized UAV Network Simulator based on ns-3 and Ardupilot," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, ACM, 2018, pp. 37–45.

[44] J. Gascon-Samson, F.-P. Garcia, B. Kemme, and J. Kienzle, "Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud," in *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, Jun. 2015, pp. 486–496.

[45] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[46] Wikipedia. (Oct. 2021). "Random waypoint model."

[47] S. Wang, Y. Zhao, J. Xu, J. Yuan, and C.-H. Hsu, "Edge server placement in mobile edge computing," *Journal of Parallel and Distributed Computing*, vol. 127, pp. 160–168, 2019.

[48] M. Peuster. (Oct. 2020). "Containernet."

[49] C. E. Rothenberg, D. A. Lachos Perez, N. F. Saraiva de Sousa, R. V. Rosa, R. U. Mustafa, M. T. Islam, and P. H. Gomes, "Intent-based control loop for dash video service assurance using ml-based edge qoe estimation," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, IEEE, 2020, pp. 353–355.

[50] C. Fiandrino, A. B. Pizarro, P. J. Mateo, C. A. Ramiro, N. Ludant, and J. Widmer, "OpenLEON: An End-to-End Emulation Platform from the Edge Data Center to the Mobile User," *Computer Communications*, vol. 148, pp. 17–26, 2019.

[51] G. Yang, X. Lin, Y. Li, H. Cui, M. Xu, D. Wu, H. Ryden, and S. B. Redhwan, "A telecom perspective on the internet of drones: From lte-advanced to 5g," *CoRR*, vol. abs/1803.11048, 2018. arXiv: 1803.11048.

[52] S. Abdelwahab and B. Hamdaoui, "FogMQ: A Message Broker System for Enabling Distributed, Internet-Scale IoT Applications over Heterogeneous Cloud Platforms," 2016. arXiv: arXiv:1610.00620.

[53] J. Gascon-Samson, J. Kienzle, and B. Kemme, "MultiPub: Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe," in *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, pp. 2075–2082.

[54] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Transactions on computers*, vol. 35, no. 11, pp. 940–948, 1986.

[55] 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.

[56] 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.

[57] O. Robotics, *Robot Operating System(ROS)*, https://www.ros.org/about-ros/, 2021.

[58] 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.

[59] L. Codecá, R. Frank, S. Faye, and T. Engel, "Luxembourg SUMO Traffic (LuST) Scenario: Traffic Demand Evaluation," *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 2, pp. 52–63, 2017.

[60] 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 International Conference on Robotics and Automation (ICRA)*, Xi'an, China: IEEE, 2021, p. 12.

[61] 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.

[62] SimPy. (2016). "Documentation for simpy."

[63] I. Lera, C. Guerrero, and C. Juiz, "Availability-aware service placement policy in fog computing based on graph partitions," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3641–3651, 2018.

[64] M. I. Naas, L. Lemarchand, J. Boukhobza, and P. Raipin, "A graph partitioning-based heuristic for runtime iot data placement strategies in a fog infrastructure," in *Proceedings of the 33rd annual ACM symposium on applied computing*, 2018, pp. 767–774.

[65] R. Deng, R. Lu, C. Lai, and T. H. Luan, "Towards power consumption-delay tradeoff by workload allocation in cloud-fog computing," in *2015 IEEE international conference on communications (ICC)*, IEEE, 2015, pp. 3909–3914.

[66] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vii, and Y. Xiong, "Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, IEEE, 2021, pp. 206–216.

[67] 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.

[68] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE*

INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 1270–1278.

[69] 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.

[70] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, “Samza: Stateful scalable stream processing at linkedin,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[71] O.-C. Marcu, R. Tudoran, B. Nicolae, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, “Exploring shared state in key-value store for window-based multi-pattern streaming analytics,” in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE Press, 2017, pp. 1044–1052.

[72] B. Confais, A. Lebre, and B. Parrein, “Performance analysis of object store systems in a fog and edge computing infrastructure,” in *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXIII*, Springer, 2017, pp. 40–79.

[73] L. Affetti, “Consistent stream processing: Doctoral symposium,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS ’17, Barcelona, Spain: ACM, 2017, pp. 355–358, ISBN: 978-1-4503-5065-5.

[74] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, *et al.*, “F1: A distributed sql database that scales,” 2013.

[75] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, “Mobile fog: A programming model for large-scale applications on the internet of things,” in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, 2013, pp. 15–20.

[76] D. M. Lewin, “Consistent hashing and random trees: Algorithms for caching in distributed networks,” Ph.D. dissertation, Massachusetts Institute of Technology, 1998.

[77] Y. Teranishi, R. Banno, and T. Akiyama, “Scalable and locality-aware distributed topic-based pub/sub messaging for iot,” in *2015 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2015, pp. 1–7.

[78] *Kafka connect*, https://docs.confluent.io/current/connect/intro.html, Accessed: 2018-03-07.

[79] *DataStax configuring data consistency in apache cassandra*, https://docs.datastax.com/en/cassandra/2.1/cassandra/dml/dml_config_consistency_c.html, Accessed: 2018-02-19.

[80] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *2014 IFIP Networking Conference*, IEEE, 2014, pp. 1–9.

[81] S. K. Monga, S. K. Ramachandra, and Y. Simmhan, "Elfstore: A resilient data storage service for federated edge and fog resources," in *2019 IEEE International Conference on Web Services (ICWS)*, IEEE, 2019, pp. 336–345.

[82] L. Epifâneo, C. Correia, and L. Rodrigues, "Cathode: A consistency-aware data placement algorithm for the edge," in *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, IEEE, 2021, pp. 1–10.

[83] J. Noor, M. Srivastava, and R. Netravali, "Portkey: Adaptive key-value placement over dynamic edge networks," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 197–213.

[84] K. Sonbol, Ö. Özkasap, I. Al-Oqily, and M. Aloqaily, "Edgekv: Decentralized, scalable, and consistent storage for the edge," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 28–40, 2020.

[85] A. W. Services. (2022). "Aws wavelength."

[86] AT&T. (2022). "Att private 5g edge to make networks smarter and faster."

[87] P. Newswire. (2022). "Edgemicro expanding footprint with 5 new micro data centers."

[88] 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*, 2021, pp. 182–196.

[89] H. Gupta, T. C. Landle, and U. Ramachandran, "Epulsar: Control plane for publish-subscribe systems on geo-distributed edge infrastructure," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, IEEE, 2021, pp. 228–241.

[90] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity: A case of knative," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 13–18.

[91] OpenFaaS. (2022). "Openfaas - serverless functions, made simple."

[92] Apache. (2022). “Apache openwhisk – open source serverless cloud platform.”

[93] E. T. S. I. (ETSI). (2017). “Mobile edge computing (mec); radio network information api.”

[94] Y. Tan, C. Han, M. Luo, X. Zhou, and X. Zhang, “Radio network-aware edge caching for video delivery in mec-enabled cellular networks,” in *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, IEEE, 2018, pp. 179–184.

[95] Y. Li, P. A. Frangoudis, Y. Hadjadj-Aoul, and P. Bertin, “A mobile edge computing-assisted video delivery architecture for wireless heterogeneous networks,” in *2017 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2017, pp. 534–539.

[96] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, “Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 2287–2295.

[97] 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*, 2017, pp. 1–13.

[98] M. Sethuraman, A. Sarma, A. Dhekne, and U. Ramachandran, “Foresight: Planning for spatial and temporal variations in bandwidth for streaming services on mobile devices,” in *Proceedings of the 12th ACM Multimedia Systems Conference*, 2021, pp. 227–240.