Doctoral Dissertation

Design and Implementation of Decentralized Smart City Services on the Edge

Jose Paolo V. Talusan

December 1, 2020

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Jose Paolo V. Talusan

Thesis Committee:

Professor Keiichi Yasumoto	(Supervisor)
Professor Kazutoshi Fujikawa	(Co-supervisor)
Associate Professor Hirohiko Suwa	(Co-supervisor)
Assistant Professor Yugo Nakamura	(Co-supervisor)

Design and Implementation of Decentralized Smart City Services on the Edge*

Jose Paolo V. Talusan

Abstract

Urban cities faced with overpopulation are embracing data-intensive applications in order to maximize constrained resources. They deploy Internet-of-Things (IoT) devices throughout the city, gathering data to be processed and analyzed in the cloud. However, the need for real-time services requires a shift from the cloud towards edge and fog computing paradigms. In addition, growing concerns for privacy, trust and autonomy require moving away from centralized approaches to a more decentralized one. This edge-centric computing delegates processing to edge-devices. A new framework for gathering, distributing, processing and aggregating tasks and results over heterogeneous devices must be created. In this dissertation, we focus on three challenges to realize such a framework: 1) how to implement it over distributed devices, 2) how to ensure service resiliency, and 3) how to deploy smart city services on this framework. For the first challenge, we describe our middleware based on the framework. We show how it is deployed over distributed nodes by implementing a workspace recognition service. We introduce workflows for data distribution and aggregation, task allocation and decentralized execution. We show that in-situ resource provisioning on distributed nodes decreases execution time by 20% for every node added. For the second challenge, we improve on the middleware and create a testbed that verifies the effects of anomaly detection and node configuration on service response times. We show that data falsification attacks can be prevented without additional burden on the system. We also show that varying distributed node configuration affects overall

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, December 1, 2020.

execution time. For the third challenge, we realize a complete smart city service on the middleware. We develop a distributed route planning service with task allocation algorithm that utilizes city road side units as distributed nodes. We explore the feasibility of the service and task allocation algorithm by measuring the trade-off between route travel time accuracy and processing time, and comparing it to a naive implementation. We show via simulation and emulation using real-world data, that our routing system with task allocation algorithm is able to process queries 50% faster with only a 7% decrease in travel time accuracy.

Keywords:

distributed computing, middleware, transportation, vehicle routing, road side units

Contents

List of Figures **vi**

List of Tables **ix**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Problem Statements 3
1.3 Organization of Dissertation 4

2 Related Literature **6**
2.1 Cloud, Fog and Edge computing 6
2.2 Internet of Things and IoT Platforms 7
2.3 Urban Middleware and Task Assignment Problems 7
2.4 Smart Mobility 8
2.5 Centralized/Decentralized Routing 9

3 Information Flow of Things Framework and Middleware **10**
3.1 Introduction 10
3.2 Literature Review 11
3.3 IFoT Middleware Architecture 13
3.3.1 Platform Architecture 13
3.4 Workspace Context Recognition Service 15
3.4.1 Service Scenario 15
3.4.2 Details of the Task graph 16
3.5 Implementation and Evaluation 17
3.5.1 Implementation 17
3.5.2 Centralized vs. Distributed Task Execution 21
3.6 Summary 24

4 Evaluating Smart City Services on Information Flow of Things Middleware **26**
4.1 Introduction . . . 26
4.2 Smart City . . . 26
4.3 Smart City Service Middleware Requirements . . . 28
4.4 Improving the IFoT Middleware . . . 29
4.5 Resilient Smart Mobility Service . . . 31
4.6 Assumptions of the Service . . . 32
4.6.1 Details of the Task Graph . . . 33
4.6.2 Resiliency . . . 33
4.6.3 RSU Location Considerations . . . 34
4.7 Implementation . . . 37
4.7.1 Testbed Implementation . . . 37
4.7.2 Service Simulation . . . 37
4.7.3 Delay Emulation . . . 38
4.8 Evaluation . . . 41
4.9 Summary . . . 42

5 Route Planning through Distributed Computing using Road Side Units as Resource **44**
5.1 Introduction . . . 44
5.2 System Architecture . . . 45
5.2.1 Spatial Region . . . 48
5.2.2 Decentralized Route Planning Service . . . 48
5.2.3 User and Query Tasks . . . 50
5.3 Distributed Route Planning . . . 51
5.3.1 Definition of the Problem . . . 51
Delay . . . 52
Accuracy . . . 54
Impact of Accuracy and Delay . . . 55
Utility Function . . . 56
Objective Function . . . 56
5.3.2 Region of Interest Heuristic . . . 56
5.3.3 Decentralized Route Planning Example . . . 59

5.4 Experiment and Results . . . 61
5.4.1 Phase 1: Feasibility Test and Parameter Identification . . . 61
5.4.2 Phase 2: Real-World Data and Scalability . . . 66
Container Benchmarking . . . 66
Experiment Parameters . . . 69
5.4.3 Experiment Evaluation . . . 70
Task Allocation . . . 72
Processing Time . . . 72
Accuracy . . . 74
Concurrent Query Count . . . 76
Route Generation . . . 78
5.5 Discussion and Limitations . . . 78
5.5.1 Discussion . . . 78
5.5.2 Limitations . . . 79
5.6 Summary . . . 80

6 Conclusion **81**
6.1 Summary . . . 81
6.2 Limitations and Future Work . . . 82

Acknowledgements **85**

References **86**

Publication List **95**

List of Figures

3.1 Paradigm shift of IoT-based systems 11
3.2 Overview of IFoT Middleware Platform Architecture 13
3.3 Task graph for workspace context recognition service 16
3.4 System architecture 18
3.5 Placement of Environmental sensors in the implementation 19
3.6 Confusion matrix for SVC model 21
3.7 Demonstration of possible output of the Smart Room context recognition service, where blank(no use), blue(low use), green(medium use) and red(high use) 22
3.8 Execution times for large sets of data in single queries with varying number of workers 23
3.9 Execution times for small sets of data in multiple parallel queries with varying number of workers 23

4.1 Smart City System Architecture 27
4.2 Improved IFoT Framework Architecture 29
4.3 Service Broker generated Task Graph for M clusters 32
4.4 Q Response to Attack 35
4.5 RSU Locations - Grid Layout 36
4.6 Injection Points for Delay Components in Simplified System . . . 39
4.7 Different possible network configurations for a system with 8 nodes. (Left) 4 clusters with 2 workers each, (Right) 2 clusters, with 4 workers each 40
4.8 Overall Execution Time vs Clustering Configurations 41

5.1 Target spatial area is divided into grids 47

5.2 Data gathering and propagation architecture of the middleware. Each RSU gathers mobility data from vehicles in their area and then propagates them to other RSUs within their search area. . . 49
5.3 Distributed Route Planning architecture of the middleware and the flow of interaction between components. Once the user sends a query to the Broker, `getOptimalSequenceGrid` generates an optimal route for the query. The query is then divided into tasks which are then sent to corresponding RSUs by `distributeTasks`. Each `findPartialRoute` and `sendPartialRoute` is run in a sequential manner. Once all routes have been sent to the broker and aggregated, the user receives the complete route. 52
5.4 Decentralized Route Planning example to show the effects of accuracy and delay 55
5.5 Region of interest based on varying Neighbor levels for RSU at the location (i, j). Neighbor level 1 includes the N0 grid at (i, j) and all N1 grids. Level 2 includes all grids N0, N1, and N2. As neighbor level increases, more grids become alternative grids. 57
5.6 Synthetic Processing time for all 12,000 queries. Neighbor Level 0 takes more than 2 times longer than when utilizing neighbor grids 63
5.7 The trade-off when utilizing neighboring nodes to decrease query response time is the increase in Manhattan Distance between optimal and allocated RSUs, resulting in a decrease in model accuracy 63
5.8 0^{th} Level Neighbors Utilization for the first 100 queries 64
5.9 2^{nd} Level Neighbors Utilization for the first 100 queries 65
5.10 Comparing the overall processing time for 1000 trip queries using Docker containers and Raspberry Pi 3B. Mac mini and Macbook Pro devices are running 49 containers simultaneously and benchmarks are run simultaneously on all containers. 67
5.11 The target area is divided into a 5x5 grid layout. Each over-utilized grid is further divided into sub-grids. The bar shows the density of road segments in the grid. 68
5.12 CDF curves showing the effect of Neighbor Levels on the query response times 70

5.13 Effect of utilizing neighbor nodes during task allocation on total query time 71
5.14 Task Allocation for 1,000 queries, each divided into sub-tasks . . . 72
5.15 Histogram of Manhattan Distances based on Neighbor Levels . . . 74
5.16 Effect of varying delay on average travel time errors. The rightmost point (Dynamic) is based on assigning different delays based on road speed changes. Nodes get speed updates in increments of 5, 10, 30, and 60 minutes based on how often the road speed changes. 75
5.17 Box plots showing the effect of the number of concurrent queries sent on the query response times of the system (Neighbor level 1) 76
5.18 Routes generated with varying neighbor levels. Circle and triangle are source and destination respectively. Only the 2^{nd} half of the route is shared by all. 77

List of Tables

5.1 List of symbols 46
5.2 Comparison of Computational Resources of Devices Used in the Experiment 69
5.3 Effect of Query Count on Allocation and Processing (Neighbor Level 1) 71

1 Introduction

1.1 Background and Motivation

The Internet of Things (IoT)-centric concepts like e-health, autonomous vehicles, smart grid and smart transportation, etc. have a ubiquitous presence now. IoT has drastically changed our society by providing services that offer seamless connectivity between man and the virtual world. By 2022, Cisco predicted that there will be 50 billion things connected to the Internet [1]. In 2020 alone, there are an estimated 17 billion mobile devices being used by more than five billion users [2]. Such devices have become such a ubiquitous part of our daily lives. As the number Internet-of-Things (IoT) devices grows rapidly, the amount of generated information as well as the computational power available. Harnessing the available data and computational resources form the basis of what we call ubiquitous computing [3]. Cities, particularly the urban setting, have been looking at this vast amount of data and untapped computational power as a way to improve their citizens' quality of life.

Transportation is one aspect of urban city life that has only gotten worse in recent years. As city populations increase, the already constrained transportation network resource is getting pushed to the limit. Cities are turning to data-intensive applications and crowd-sourced information to alleviate this problem. Companies such as Google, Apple, and Waze have deployed applications that allow users to utilize crowd-sourced data and their devices' computational resources, to help traverse the city roads. While these services are able to provide what they promise, highly efficient planned trips and routes, they rely heavily on cloud computing architectures. These centralized platforms need to access remote servers, via the Internet, which hold all the data and performs processes.

These centralized, cloud-based solutions while efficient, exposes both users and

their data to privacy and vulnerability issues. The issue of privacy, which has always been a central topic in academic circles, has recently been thrust into the limelight [4]. People have become aware of the fact that some companies who have access to their data have sold it for profit [5]. Data stored remotely are more exposed to data leakage and privacy attacks. By relying on centralized services in remote data centers, cities also risk service availability issues during disaster scenarios, precisely when such services are of most importance. In addition, the distance between data sources and remote data centers, introduce latency. This is satisfactory for current applications in which latency is not too great a factor for each request. However, latency demands of future technologies such as autonomous vehicles expose the limitations of centralized cloud-base route planning models. The need for *real-time* service has started a shift from cloud-based services towards edge and fog computing paradigms [6] [7]. However, utilization of computational resources and the use of resource constrained devices are not taken into consideration.

The number of computing devices—and thus computational power— available at the edge is growing rapidly; this trend is projected to continue in to the future [8]. This is being viewed as an opportunity to offload computing from the cloud to edge devices. By linking these devices together in a private sub-network, a reliable and secure network can be created for smart city services that can remain in operation even without connection to cloud services. These networks can take advantage of the constant stream of data being generated to provide regional IoT services to its users.

In response to these trends and limitations of current technologies, a new framework must be established to handle these constant streams of data. A new edge-centric [9] framework for gathering, distributing, processing and aggregating tasks, must be realized. This framework should allow for real-time processing and analyzing of data streams through distributed computing by edge devices. The Information Flow of Things (IFoT) [10] is one such framework.

In this study, we aim to realize such a framework, implement it on low-power and resource constrained devices, and deploy smart city services on it. Before this framework can be realized, challenges must be addressed and solutions evaluated for their efficiency.

1.2 Problem Statements

In order to implement the IFoT framework, we must first describe design the architecture. We must then solve inherent problems which arise when dealing with distributed devices. The middleware that arises from this framework as well as the services that would be deployed must be evaluated in order to prove its efficiency. We identify these challenges to be solved in this dissertation and we organize them as follows:

Challenge 1: How is this framework designed and implemented on distributed devices?

We investigate preliminary literature for the IFoT framework and design our own architecture based on its assumptions and requirements. We focus on functionalities that such an architecture should have. (1) It should be deployed over heterogeneous distributed devices. (2) It should be able to store and process data in a timely manner without the use of Cloud-based computing systems.

We tackle this challenge by reviewing the framework proposed by [10] as well as other preliminary investigations by [11]. We then design a middleware based on this IFoT framework, detailing each aspect of the system that would be deployed over distributed devices. Finally, we present an implementation based on this middleware. It is a platform meant to utilize in-situ distributed processing that was designed to function without the need for Cloud-based services.

Challenge 2: How to ensure resilient and timely smart city services over this framework and how do we measure its efficiency?

We improve upon the prior system and further develop the IFoT middleware. This iteration of the middleware incorporates decentralized services with the goal of accomplishing real-time stream data processing while maintaining resiliency over distributed nodes.

In order to deploy resilient and timely smart city services, the system must be able to satisfy the following requirements: (1) Node layout and network should prioritize transmission delay and data security. (2) Services should be able to deal with data falsification attacks. (3) There should be a method to verify the efficiency of such configurations.

We solve these problems by improving upon the prior middleware we created.

(1) We measure the network and transmission delay between different configurations of brokers and workers. (2) We implement anomaly detection over the service that deals with data falsification attacks. Finally for (3), we build a testbed that allows us to verify the effects of these on the service.

Challenge 3: How can we realize distributed smart city services on this middleware?

The middleware should be able to accommodate a varied assortment of smart city services. (1) First, the service should be deployed over decentralized nodes that communicate via our middleware. (2) Then, the service should provide an algorithm that highlights and utilizes the decentralized nature of the middleware. (3) Finally, the service should be able to respond to the user's query within acceptable times.

We solve this by creating a middleware where services can be deployed in a seamless manner. We utilize containers such as Dockers to be able to containerize services and simplify service deployment. The middleware must then be able to utilize the distributed nature of the system with little to no additional configuration from the service creators. To verify this, we improve the middleware and deploy a route planning service over the middleware in a real-world environment. (1) First, the service should be deployed over decentralized nodes that communicate via our middleware. (2) Then, the service should provide an algorithm that highlights and utilizes the decentralized nature of the middleware. (3) Finally, the service should be able to provide routes to users by planning and generating routes over these decentralized nodes.

By making sure that the middleware is able to handle generic services, it would be easy to extend the middleware for multi-modal route planning or smart tourism services.

1.3 Organization of Dissertation

The rest of this dissertation is organized as follows: we present a review of related literature in Chapter 2. In Chapter 3, we discuss more about the foundation of the research, a framework called Information Flow of Things and present a pre-

liminary approach to implementing this framework. In Chapter 4, we improve on this approach by adding smart city services that run on the middleware as well as create a test-bed to measure its effectiveness. In Chapter 5, we further evaluate the system by creating a distributed route planning service, adding a task distribution algorithm that decreases the overall processing time, and incorporating real world data to our tests. Finally, we present our conclusions in Chapter 6.

2 Related Literature

In this chapter, we present a review of studies and discuss the concepts related to our study.

2.1 Cloud, Fog and Edge computing

Due to the large amount of data produced by billions of IoT devices, traditional centralized cloud servers will eventually face the problem of non-negligible delays when providing IoT-related services. Edge [12] and Fog [6] computing are approaches to mitigating the reduced quality and increased service costs of cloud computing. Edge and fog computing are both “edge-heavy computing" paradigms where data processing is executed on components in or near the data source.

Edge computing may act as a bridge between IoT devices and the cloud. Edge and fog computing make it possible to minimize the latency of tasks compared with the cloud. These platforms perform roles such as IoT device management, network management and data processing and transferring. While edge computing is able to minimize latency as well as efficiently use available bandwidth, it still faces challenges with regards to data partitioning and offloading of tasks. In addition, the demerit of these approaches is the investment needed to replace such network constituents like Information-Centric Networks (ICNs).

Typical implementations of edge computing still rely on working with offsite processing centers or cloud-based services in order to offer their services. This balances the best of the two implementations, data collection and aggregation are done on the edge while computationally intensive processes are done in the cloud. The edge devices augment the cloud by performing simpler tasks that help offset the communication delays. Only recently have processes been completely executed on the edge [13] rather than as a combination of cloud and edge devices.

2.2 Internet of Things and IoT Platforms

Many IoT platforms have been designed and implemented to interconnect IoT devices and process data. Axeda * and Arkessa † are some of the platform as a service (PaaS) cloud-based architectures. However as with all cloud-based services, these platforms require a constant connection to the Internet. The same applies to edge [14] and fog computing [6] paradigms, which move the execution of data processing closer to the data source. While these paradigms are much quicker than traditional cloud computing, this architecture will still cause large delays in providing services as well as waste resources on the cloud and network.

2.3 Urban Middleware and Task Assignment Problems

The challenges associated with edge and fog computing are primarily associated with usability, coordination and task assignment. Task scheduling in fog computing with the goal of optimizing resources to minimize tasks completion time [15] and efficiently utilizing resources to improve the performance of IoT services in terms of response time, energy, and cost reduction [16] have been studied. Therefore much research has been done on urban middleware designed to coordinate large systems of heterogeneous edge or fog networks [17], [18] and [19].

A primary goal of urban middleware is to formalize how best to assign computation tasks to available resources. We refer to this problem as the task allocation problem [20]. This problem has been extensively studied in the cloud [21], [22], [23]. The main purpose of research such as this is to adaptively provide the processing resources while meeting the deadline for all jobs while taking into account running costs. The task allocation problem in cloud computing therefore typically does not take into account data transfer delay, as typically the networking between virtual machines in a cloud environment is negligible. A key component of urban middleware is resource discovery, which is the method by which edge and fog networks are identified [24], [25].

*http://www.axeda.com/

†https://www.arkessa.com/

Therefore, recent research in the provisioning of resources in edge and fog networks has become increasingly important. Skarlat et al. [26], [27] proposed a platform centered around the idea of fog colonies (sets of fog nodes) with a centralized cloud for additional resources when needed. Xu et al. [28] proposed a platform for location-based and latency sensitive applications which use micro data centers on the network edge or large centralized cloud for processing. This research includes a cloud component for additional processing and storage. Research on in-situ edge IoT devices [11] [29], works assigning tasks without relying on cloud resources.

2.4 Smart Mobility

The edge and fog computing paradigms have been leveraged to process and visualize data from sources such as road side units (RSU) and Vehicular Social Networks [30] to provide services such as Intelligent Transportation Systems (ITS) already present in Japan [31]. These are able to provide users with real-time wide-area traffic congestion information. One such application, SpeedPro [32], uses GPS location data fused with historical data to provide more reliable urban traffic speed estimates.

While Edge and Fog computing are promising for these applications, a number of challenges still exist that must be addressed. Eisele et al. [33] state that one challenge is to be able to provide a stable application environment despite the dynamism, heterogeneity, and increased failure potential of computing resources at the "edge" away from data centers.

Security and privacy [34] are also points of concern. Due to their reliance on spatio-temporal data, measures need to be taken in order to preserve data integrity and to detect anomalies within such systems [35]. A large focus of research in this field is on implementation of sensor systems for transportation, communication and infrastructure monitoring [36], [37], [38], [39], [33]. Traditional anomaly detection in this context is based on classification, statistical, state based, clustering or information theory [40]. Classification methods are usually based on Support Vector Machines (SVM), Bayesian Models, Gaussian Processes or Neural Networks [41].

2.5 Centralized/Decentralized Routing

Dijkstra [42], Bellman [43] and Ford [44] proposed some of the first routing planning algorithms. Routing algorithms such as A* [45] use heuristics to guide the shortest path search while contraction hierarchies [46] simplify the graph for faster search.

Current state of the art route planning is typically deployed in centralized cloud systems [46], [47], [48]. In this architecture the routing algorithms are deployed in a central location from which it serves user queries. Within this context QoS improvements (e.g., in terms of query response time) have been made by parallelizing shortest path algorithms [49], [50], [51]. These parallelized algorithms split processing over multiple nodes. These approaches provide high scalability, optimal for cloud-based services. However, these models assume a shared memory and do not take into account network latency between nodes, and therefore are not easily adaptable to edge or fog centric architectures.

3 Information Flow of Things Framework and Middleware

In this chapter, we lay the groundwork of our study. We describe the *Information Flow of Things* framework. We design and discuss the architecture needed to create our middleware based on this framework. We demonstrate an initial implementation of this middleware and evaluate its capabilities.

3.1 Introduction

Cloud computing is currently the main platform used for deploying IoT services [52]. However, cloud-based approaches is not a catch-all solution for all types of environments and services. Locations with limited or no access to the Internet will have a difficult time utilizing cloud-based services. While services which offer a level of privacy and resiliency will find the dependency on remote servers inadequate for their needs.

Now edge and fog computing paradigms are attracting attention with their ability to process data much closer to the source. In edge-based existing studies [53], [54], edge clouds are deployed and used in a metropolitan based environment to process tasks with low delay. However utilization of computational resources and use of resource constrained devices are not taken into consideration.

In response to these trends and limitations of current technologies, a new edge-centric [9] framework must be realized. This framework should allow for real-time processing and analyzing of data streams through distributed computing by edge devices. The Information Flow of Things (IFoT) [10] is one such framework.

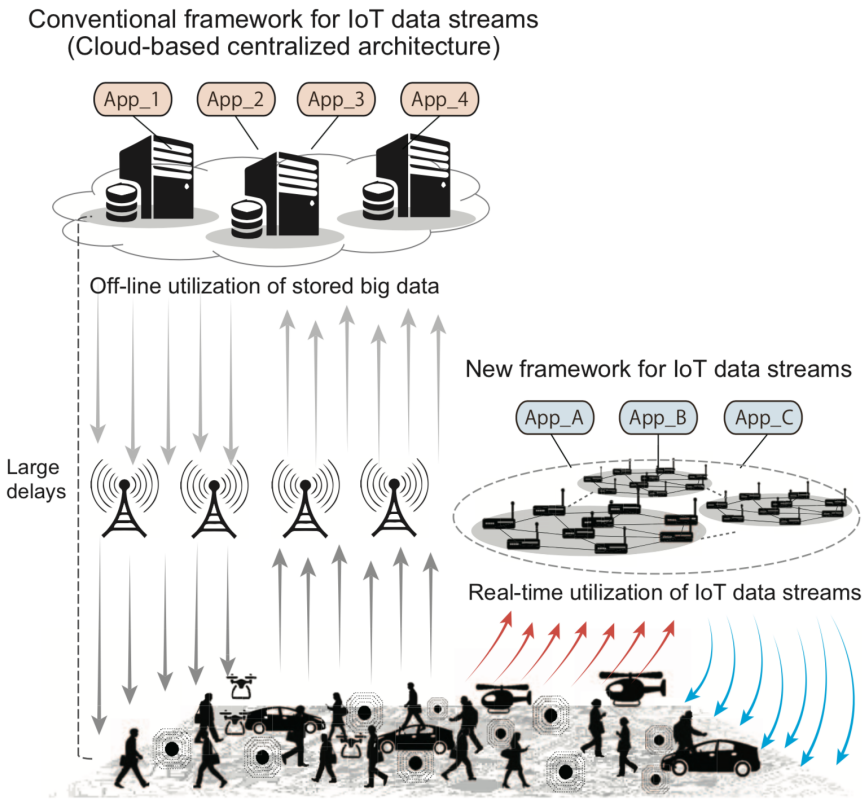

Figure 3.1: Paradigm shift of IoT-based systems

3.2 Literature Review

Information Flow of Things (IFoT) is a proposed framework aimed towards processing massive IoT data streams in real-time manner by edge servers and IoT devices [10]. It is designed to provide delay-aware services through mechanisms such as in-situ distributed computing and data aggregation. It aims to have a better cost-performance index than cloud-based and edge-based approaches [11]. The goal is to achieve an improved satisfaction level for delay-sensitive applications (such as smart city or smart mobility) while being able to aggregate user data in a secure and timely manner with a certain level of robustness against privacy and security threats.

While distributed computing through IoT devices is an attractive prospect, there is still the challenge of managing and deploying them over heterogeneous IoT

devices. Work is still being done to address where current frameworks fall short in dealing with the heterogeneity of distributed computing. Tasklets [25], [55] and Bhave et al. [56] attempted to ease the burden of heterogeneity for distributed and edge computing by using middleware and virtualization technologies to efficiently handle multiple heterogeneous devices and tried to pool their computation resources together.

While studies on containerization and middleware platforms have been found feasible and successfully done on commodity devices such as Raspberry Pi [57], [58], [59], and have been able to provide some form of distributed processing, these systems do not focus on providing service for users within an area. Also, these prior systems and platforms while being able to create a middleware on heterogeneous devices, were not utilizing the computational resource of the pooled devices for more sophisticated data processing/analysis by means of distributed machine learning.

In [11], the Information Flow of Things framework was proposed. A preliminary architecture was presented and implemented. The middleware was tested using limited use-cases. In this dissertation, we extended this framework for actual smart city services. We also created a testbed, which emulated distributed edge devices using Docker containers and real-world data, to further evaluate our improved framework. In addition, we implemented a real-world smart city service use case to further demonstrate effectiveness of our system. The prior authors also presented an in-situ resource provisioning with adaptive scale-out algorithm for regional IoT services improves the quality of service (QoS) of delay-sensitive IoT services. However, their algorithm cannot be implemented for our use-case and so our implementation features a new resource constrained optimization algorithm that is made specifically for route planning services.

Figure 3.1 shows the paradigm shift that IFoT hopes to accomplish. It aims to serve as an alternative to conventional IoT frameworks when dealing with IoT data streams.

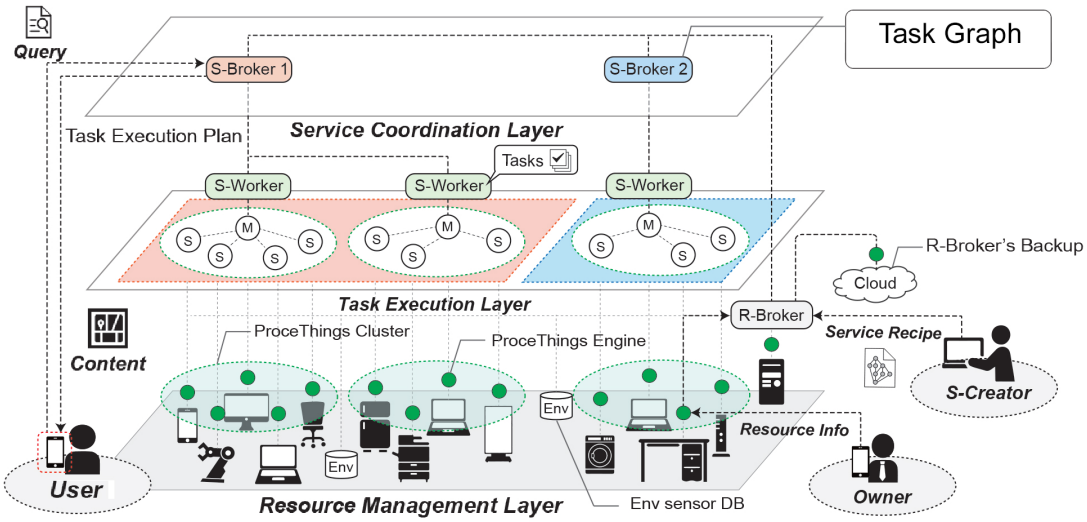

Figure 3.2: Overview of IFoT Middleware Platform Architecture

3.3 IFoT Middleware Architecture

In order to realize this framework, we have to implement a middleware that would run on devices at the edge. The IFoT middleware platform (parts of its mechanisms are presented in [11]) is a concrete realization of IFoT framework. In this section, we extend upon the previous work on the IFoT middleware done by Nakamura *et al.* [11].

The middleware consists of three main layers: *Resource Management Layer, Task Execution Layer and Service Coordination Layer.* All IoT devices in the platform are considered nodes and they may serve different functions within the architecture shown in Fig. 3.2.

3.3.1 Platform Architecture

- *Resource Management Layer*: manages IoT devices that participate in the platform. It consists of the *resource broker (R-Broker).* It is manually set by the community as (typically) the most powerful node in the network and has information on all available nodes. It also manages all the nodes in the platform.

  The R-Broker handles resource registration by resource owners through

a web interface. The owners provide information of IoT devices such as processing capabilities, available sensors and location details to the resource broker. The registered nodes are then configured into Docker nodes, to be configured into either *Service Brokers* or *Service Workers* depending on their computational capability, comprising the service coordination layer and the task execution layer explained below.

- *Service Coordination Layer*: handles the communication between end-user and the task execution layer. It consists of the *service brokers (S-Brokers)*. The S-Brokers manage services. Each S-Broker is the gateway by which users query the service through a web interface. S-Broker is assigned manually by a service creator to (typically) the most powerful node available after the R-Broker.

  S-Broker manages multiple S-Workers within its service area and adds more S-Workers/worker nodes to its manageable resource pool to provide needed QoS level for the current computation demand (i.e., the number of queries per unit of time) [11].

- *Task Execution Layer*: handles the execution of services or task graphs that the platform offers the users. It consists of *service workers (S-Workers)*. This layer is configured to function as a cluster and is designed to execute tasks in a distributed manner.

Once nodes are registered into the platform, they are setup as clusters for a specific location. *Clusters* are the task execution block of the platform which are composed of the following:

- Service Worker (S-Worker): is the virtual representation of the task execution cluster. It is formed by a Docker Swarm cluster which consists of a master node and multiple worker nodes. They are in charge with communicating with the S-Brokers regarding the user queries for task execution. Upon receiving requests, they are the ones that manage the task distribution to multiple worker nodes.

- Master Nodes: handle the task distribution to the multiple worker nodes.

Databases such as the environmental sensor databases, are placed on master nodes as well. In our implementation these are deployed on Raspberry Pi.

- Worker Nodes: are the basic execution nodes of the platform. They are tasked with executing tasks allocated by the master node. Each worker node is a single IoT device with limited or constrained computational resources. In our implementation worker nodes are Raspberry Pi.
- Environmental sensor database (envDB): is time series database that collects and aggregates data from the sensors connected to the platform. Services such as activity recognition will gather data from envDBs. These are assigned to the master nodes of S-Workers. In this implementation of the middleware, centralized storage is used.

3.4 Workspace Context Recognition Service

In this section, we present a workspace context recognition service scenario as a typical use case for the IFoT middleware platform. We give our assumptions for this scenario and discuss the task graph that details how queries are handled.

3.4.1 Service Scenario

We assume that future smart offices will have many *free address workspaces* where many environmental sensors are installed as well as having their own network infrastructure. Employees can freely use these rooms for meetings, work, and recreation at anytime. However, if the office space is too large, it is difficult for employees to figure out which workspace is currently available and suitable for their needs. This is our motivation to develop the workspace context recognition service.

Workspace Context Recognition Service: Using this service, office employees can get useful information regarding the room context such as the comfort level, noisiness, and the possible over use or under use of certain rooms. This information is generated through the data processing (Statistical processing and machine learning inference using pre-trained models within the platform) of data from environmental sensors which are located in the rooms. This service can be accessed

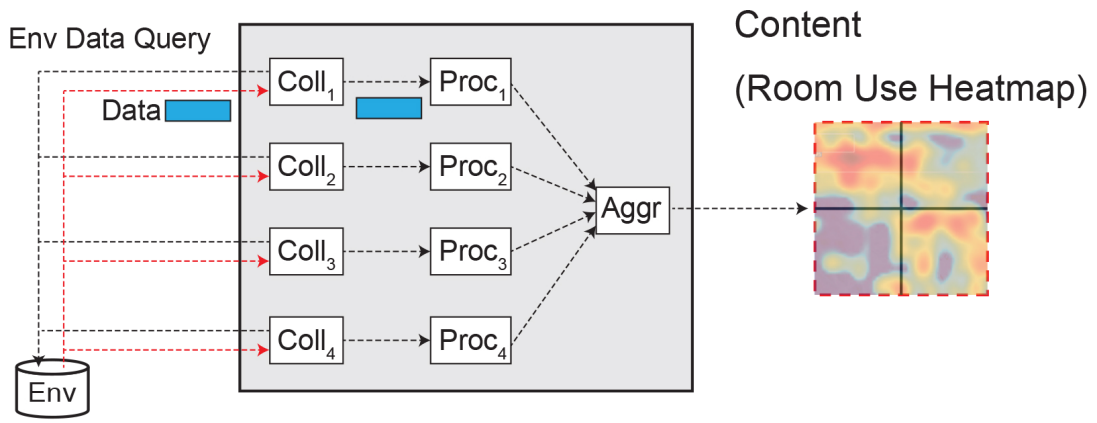

Figure 3.3: Task graph for workspace context recognition service

by the employees through the S-Broker on the local intranet. All information remains private since data is only stored locally within the nodes.

The number of monitored rooms and the number of people querying the status of these rooms, affect the performance of the platform. Performance can be increased by adding more IoT nodes into the system, increasing the local computation resource, thus avoiding the need for a more powerful central terminal. Clusters of commodity single-board computers such as Raspberry Pis are more than enough for this application.

Furthermore, having more nodes within the platform allows the creation of more services that may be needed by employees in the office space. Addition of a service is simply a case of adding the required sensors and a task graph that details the collecting, processing and aggregating tasks.

Other use cases: This framework can be generalized to similar use cases which feature characteristics such as: geo-spatial sensor data, machine-learning, heatmap, etc. Modifications to the task graph allow the framework to handle such use cases, without the need for large changes of the entire framework.

3.4.2 Details of the Task graph

Task graphs (or service recipes) detail how a service handles queries by users. Each service has a corresponding task graph. The task graph for the workspace context recognition service is shown in Fig. 3.3. Sensor data is stored inside an

envDB located inside the monitored room. Upon receiving a query, the S-Broker, executes a task graph that is executed by the S-Worker(s).

The task graph makes use of Redis *, an open source in-memory data structure that functions as the main queueing system of the IFoT middleware. Tasks are queued onto Redis and then worker nodes monitor these queues and process them when they are free.

- Collecting Task: For the current experiment, the query includes time information for the test room. This time information is then sent to a worker node which in turn collects the data from the envDB inside the room. This task is executed in parallel for each received query. The worker obtains the data in the form of a JSON string which is passed onto the queue for processing.

- Processing Task: Processing tasks will be done in parallel, depending on the number of available worker nodes that monitor the queue. Upon receiving the JSON string from the queue, it converts these to a Pandas dataframe. It also loads the pre-trained classification model that is stored in each worker node for use. It will then use the loaded model to classify the status of the room based on the sensor data. The worker node then sends the classified labels back to the queue for the aggregator.

- Aggregation Task: This task is usually done on one worker node. It will wait either for all nodes to finish or set a timeout. Upon gathering all the coordinate and label information, it will generate a heatmap that specifies the usage of a particular area. This is then saved as an SVG image and then sent back to the S-Broker for displaying back to the user.

3.5 Implementation and Evaluation

3.5.1 Implementation

The smart room context recognition service with the IFoT middleware platform is built using Raspberry Pi 3 Model B (Pi) and Omron 2JCIE-BL01 Environmental

*https://redis.io/

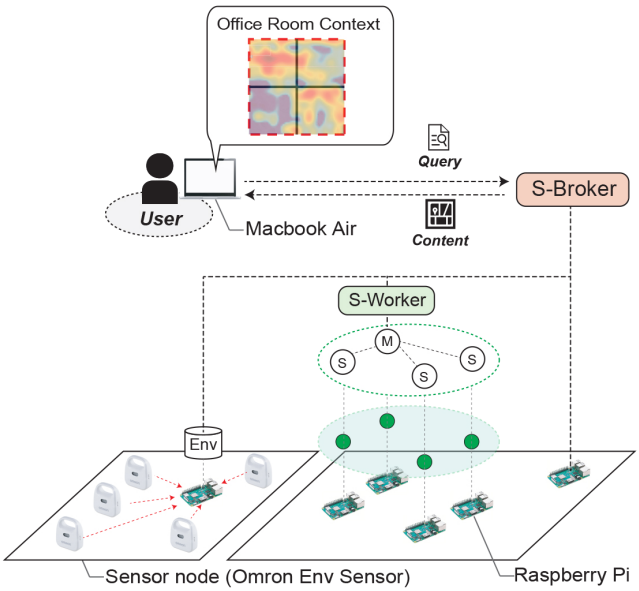

Figure 3.4: System architecture

Sensor. The Pi is equipped with 1.4GHz ARMv8 processor, 1GB DDR2 SDRAM, Wifi and Bluetooth low energy (BLE) connectivity. It uses Raspbian operating system, a version of Linux Debian, optimized for ARM. The sensor is a wireless sensor that is equipped with 7 different sensors: temperature, humidity, light, UVI, absolute pressure, noise and acceleration.

The Omron sensor measures data at 300 second intervals (this period can be set between 1 second and 1 hour), at this rate lifetime of the battery is 3 months. Each period, the sensor obtains the room's current temperature, relative humidity, ambient light, UV index, pressure and sound noise. All these information are sent to the Pi node, which listens to the sensor beacons via Bluetooth Low Energy. It receives the sensor information as well as timestamp, RSSI, sensor MAC address, gateway address, estimated distance via the RSSI, heat stroke factor, discomfort index and battery level, which is stored as a row with 18 columns in the envDB.

Figure 3.4 shows the architecture of the system. The sensor nodes were connected to a single Raspberry Pi that collects data and stores them into an environmental database. This database is accessed by a collection of nodes set in

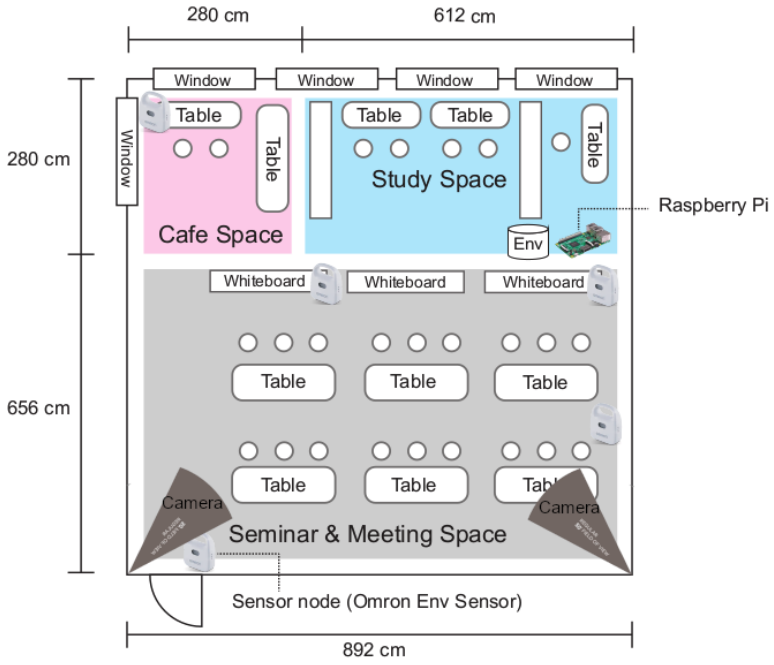

Figure 3.5: Placement of Environmental sensors in the implementation

manager/worker configuration. These nodes process and analyze the data based on the query of the user. User queries are sent to the service broker which itself is another Raspberry Pi that allows communication between clients and the service. Five environmental sensors were placed in a large multi-function room used for seminars, meetings, recreational activities, and discussions. Due to the room's size, it was further broken down into several areas as shown in Fig. 3.5.

The locations of these sensors were chosen to maximize the amount of data being collected on the varying use of the room throughout the day. Data is broadcast by the sensors every 5 minutes and were received by a sensor node (Raspberry Pi) located in the same room. This sensor node is equipped with an envDB for storing the time series data generated by all the sensors. Transmission of data is through Bluetooth Low Energy and thus RSSI information was also recorded.

Two Raspberry Pi Camera Module v2 cameras were setup in two corners of the room, as shown in Fig. 3.5, to capture ground truth data during the experiment. We then manually label each 10 minute interval based on the number of people

present.

We use 4 classification levels: **No Use, Low Use, Medium Use, High Use**. We set these based on the number of people present in the room. **0: [No Use], 1-3: [Low Use], 4-9: [Medium Use] and 10+: [High Use].** This was in an effort to keep training and classification simple since the experiment location was a single multi-functional room. Subsequent experiments with multiple rooms may increase the classification to take into account more classes.

The classification model is trained using a Support Vector Classification (SVC) algorithm via Scikit-Learn package. The features that are used from each sensor are humidity, light, noise, RSSI, and temperature. The other three features, acceleration, UVI and pressure showed little to no changes for the duration of the experiment and were not used.

Since all 5 sensors were placed in a single room at different locations, we used each sensor's 5 different sensor data as columns. Raw sensor readings were used without further modification for the data set. This results in a 26 column data set (including the timestamp which is used as a feature) with almost 2000 rows for 4 days of data gathering. This was trained offline using the SVC algorithm. The resulting machine learning model which has an accuracy of 62% as shown in Fig. 3.6, is then saved into a file for distribution to the S-Workers.

In our implementation, the model is sent first to the S-Broker and is then automatically distributed to the S-Worker via Python script. S-Worker, located outside the room simply for debugging purposes, is connected to the university's network via wired connection. The sensor node, which contains the envDB, is connected to the same network via WiFi. The sensor node can also be a worker node of the S-Worker, however for simplicity, it was configured separate from the S-Worker. In future experiments, this will be made part of the S-Worker. S-Workers and S-Brokers connect to each other through the same network above, via wired connection. The user can access the S-Broker through any method (wired/wireless, smart phone or PC) as long as they are connected to the local area network of the university. In this case, each worker node receives a copy of the same model. Upon receiving a user query, the S-Broker, executes a task graph as shown in Fig. 3.3.

The classification process is then executed in the following manner: (1) Upon

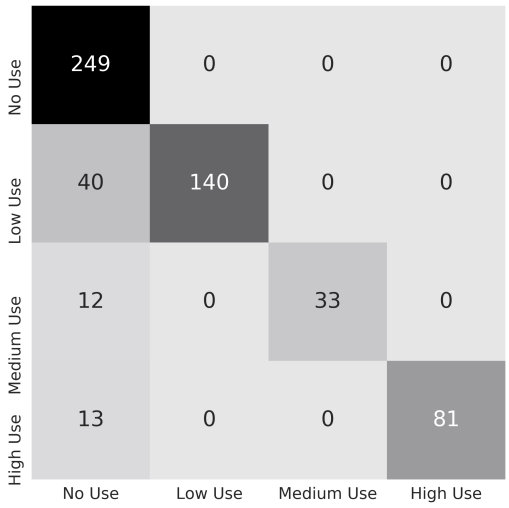

Figure 3.6: Confusion matrix for SVC model

receiving service requests, the S-Broker divides it into singular room queries. (2) It obtains and forwards the time information of each room query into the queue. (3) The S-Worker monitors this queue and assigns the tasks to a free worker node under it. (4) The worker node performs the collecting task and then using the previously received pre-trained model, the processing task. (5) Upon classification, it then sends the labeled data as well as other room information data back into a queue. (6) Again, the S-Worker, monitoring the queue, assigns this to a free worker node to perform the aggregation task as detailed in Sec. 3.4.2 The next section discusses the evaluation of this task execution and the effects of the S-Worker on the QoS.

3.5.2 Centralized vs. Distributed Task Execution

Given the setup above, a use case was imagined for the service: employees want to know information on the workspaces in the building. They query the S-Broker for information based on the sensors deployed in each space. The number of rooms or number of other entities (e.g., other building staff, local fire department monitors, etc.) regularly performing such a query at the same time may vary, leading to scenarios that require a serviceable quality of service (QoS) from the

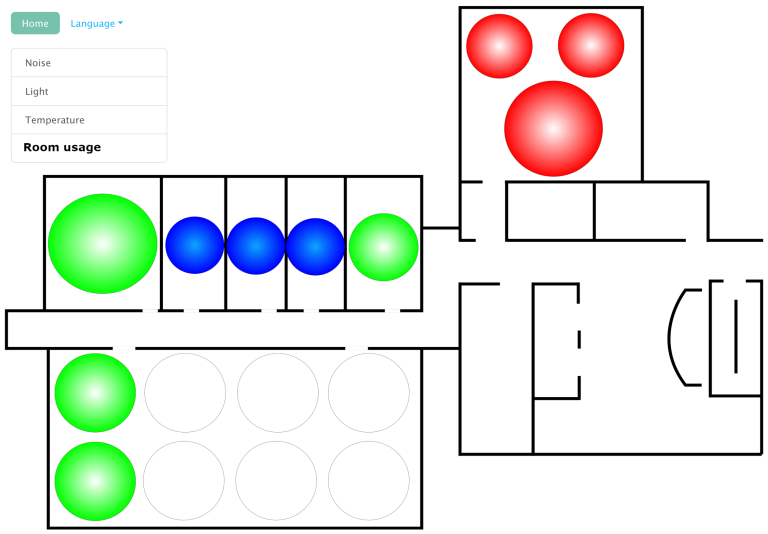

Figure 3.7: Demonstration of possible output of the Smart Room context recognition service, where blank(no use), blue(low use), green(medium use) and red(high use)

platform. The output of a user query is a corresponding label for the room they are querying, in the future this output can be displayed in the form of a heat map as shown in Fig. 3.7, where room usage is shown in various colors.

Since we only have sensors placed in a single room, we simulate how the system would behave when multiple rooms are being queried at once. To be able to do that, we randomly select 100 data points (i.e., 100 samples) from the data set and set it as the target of query.

Since the sensors would be the same regardless of the room which it is placed in, we then suppose that each row (a data point) in the data set is a different room. Based on the study of Egger et al. [60], there is a direct relationship between delay and dissatisfaction, with this we aim that the workspace context recognition service should be able to return a response within around 2 seconds.

We perform the following experiment to investigate the QoS the platform is capable of delivering. We test the system's ability to handle 100 rooms being queried at once. We first configured the platform such that the S-Worker would only run 1 worker node and then increase the number of nodes via the scale-out method detailed in [11].

Figure 3.8: Execution times for large sets of data in single queries with varying number of workers

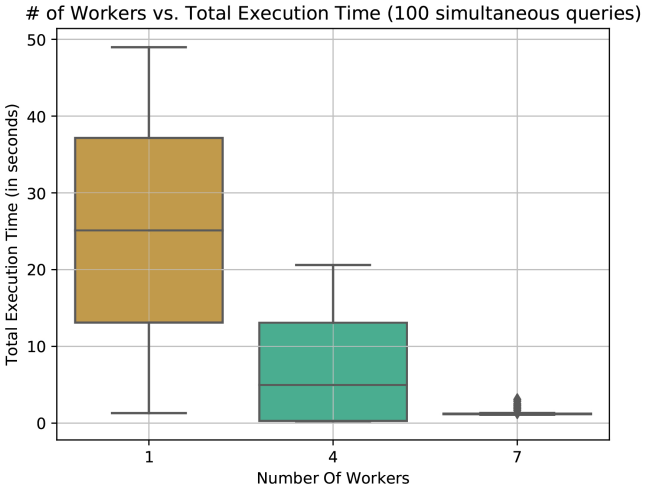

Figure 3.9: Execution times for small sets of data in multiple parallel queries with varying number of workers

We consider the total execution time as the time measured from when the user sent the query until the response of the heatmap is received by the same user. Fig. 3.8 shows the QoS of the platform against the number of worker nodes present in an S-Worker. Given large sets of data bound in a single query, a single node on average, total execution time goes beyond the set limit of 2 seconds and fails to achieve acceptable QoS. Increasing the number of nodes to 5, allows the platform to respond with an average of 2 seconds for large data set queries. This total execution time only goes lower the more nodes are added.

Of particular note is the total execution time at 7 workers. While this number is low when compared to the global average, it increases by a non-negligible amount compared to when using 6 workers. This is due to the trade-off between increasing total computation power and data communication delay. As the number of worker nodes increase, the greater the more data must be communicated between them. This diminishing return of processing speed is an interesting topic to tackle in the future and requires additional testing to completely verify the cause. One possible solution is to create improved task scheduling algorithms that assign tasks with the aim of maximizing a utility or objective, while balancing the 2 parameters, computation power and communication delay.

Next we simulate the effect of in-situ resource provisioning with scale-out [11], an additional implementation for the IFoT platform. We overload a single node using the same 100 rows used in the previous experiment, but we divide the 100 rows into individual queries and then send simultaneous queries to S-Broker. As seen in Fig. 3.9, using a single node, it has an average total execution time of 25 seconds. With an initial 4 nodes, we can decrease this total execution time to 5 seconds, quicker than a single node but still unable to meet the QoS. Finally, implementing in-situ resource provisioning to the nearest 3 neighbor nodes, we can decrease the total execution time to 1.2 seconds.

3.6 Summary

In this chapter, we designed and developed an IFoT middleware platform based on the IFoT framework that was presented in [10]. We presented a smart room context recognition service which we implemented over the IFoT middleware.

Users query the service in order to identify the usage of workspaces in an office environment. The main goal for the service is to be able to provide a response to multiple user queries within 4 seconds. We achieved this level of QoS by using the IFoT middleware platform that uses pre-trained machine learning algorithms and computational resources at the data source to classify and recognize room usage using environmental sensors. Additionally, we implemented distributed processing on the platform, this improves the QoS of the system from a total execution time of 4.2 seconds to less than 2 seconds. Furthermore, with the implementation of adaptive in-situ resource allocation, we show that we can further improve the total execution time for 100 simultaneous requests from 4 seconds to 1.2 seconds. The platform was implemented on a single room in the essence of saving time, but we show that this system is feasible and is able to deliver acceptable QoS regardless of the number of rooms being monitored.

4 Evaluating Smart City Services on Information Flow of Things Middleware

4.1 Introduction

Here we build upon our work in Chapter 3. We previously considered a middleware based on the Information Flow of Things (IFoT) framework. We explore this idea further by applying it on smart cities. In this chapter, we describe what we assume a smart city to be. We discuss what a middleware for smart city services, specifically smart transportation, require. We also consider the need to create test beds in order to verify the efficiency of our services running on the middleware.

In the rest of the chapter, we present the changes to the IFoT middleware, we then present the two services we have created along with the assumptions and system components. Finally, we evaluate our smart city service implementations.

4.2 Smart City

We assume that smart cities are equipped with sensors and computational resources that allow it to monitor and optimize their resources to maximize its services to its citizens. Fig. 4.1 shows our idea of a smart city. We assume that traffic lights and lamp posts, equipped with road side units, exist along the city roads and highways. We assume that mobility data, generated by passing vehicles, are sent to sensors nearby or within the RSUs.

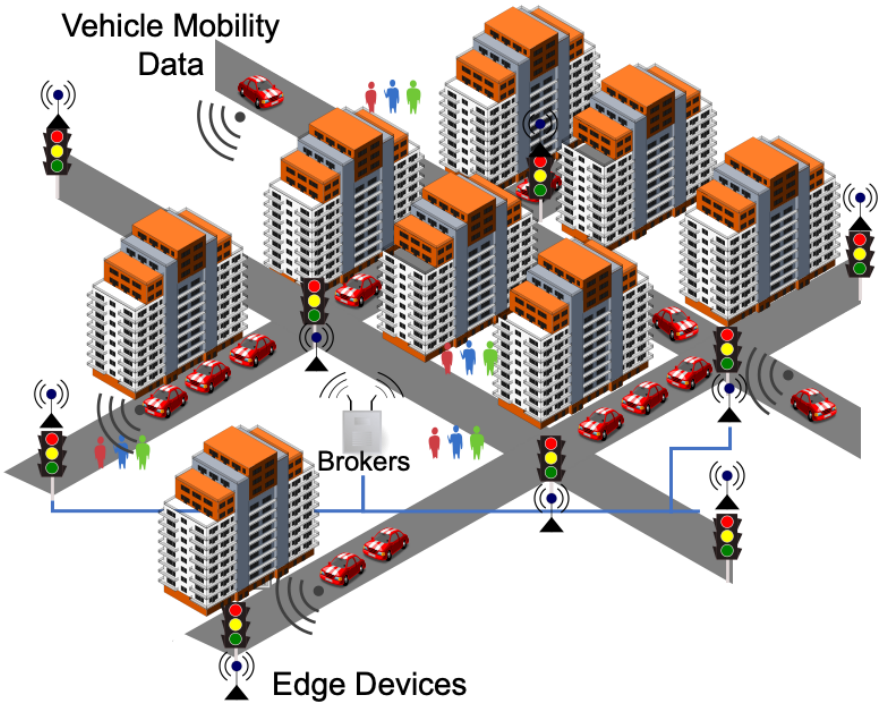

Figure 4.1: Smart City System Architecture

4.3 Smart City Service Middleware Requirements

Smart city services and applications are regarded as one of the major drivers for large scale IoT deployments that has a potential to improve the quality of life for citizens in smart cities [39]. Researchers have identified 6 main areas that can be targeted by smart cities: economy, people, governance, mobility, environment and living [61]. Each of these areas have varying requirements on middleware solutions. However, there are certain features or functional requirements [62] that a middleware meant for smart city services must meet.

Architecture: Service oriented middleware should implement a multi-level decentralized system architecture, consisting of several entities. These entities form the heterogeneous environment that the middleware will be deployed on. These can be edge devices with limited available resources or a cloud infrastructure that allows for task offloading. This is discussed was addressed by Challenge 1 in Chapter 3.

Latency and Quality of Service: IoT/IFoT environments are highly dynamic and the data and information flow is continuous. To satisfy both user and application needs, service-oriented middleware must ensure a low latency of data processing. It must be able to support near real-time delivery/processing of real time data.

Reliability: Reliability aims to increase the success rate of service delivery. The underlying communication within the middleware must be reliable since unreliable perception, data gathering, processing can lead to long delays and loss of data. The same can be said for data integrity. Services must be able to maintain and verify that data is not being manipulated to be able to make sure that responses to user queries is accurate. Both latency and reliability will be discussed in this Chapter as a solution for Challenge 2.

Data Processing Support: Information flow or streams can come from a number of sources within the IFoT framework and architecture. Sensors produce huge amounts of data which needs to be processed and provided to end-users often as an aggregated output. The middleware should support the processing of these data streams in an efficient manner, given the resource constraints of its

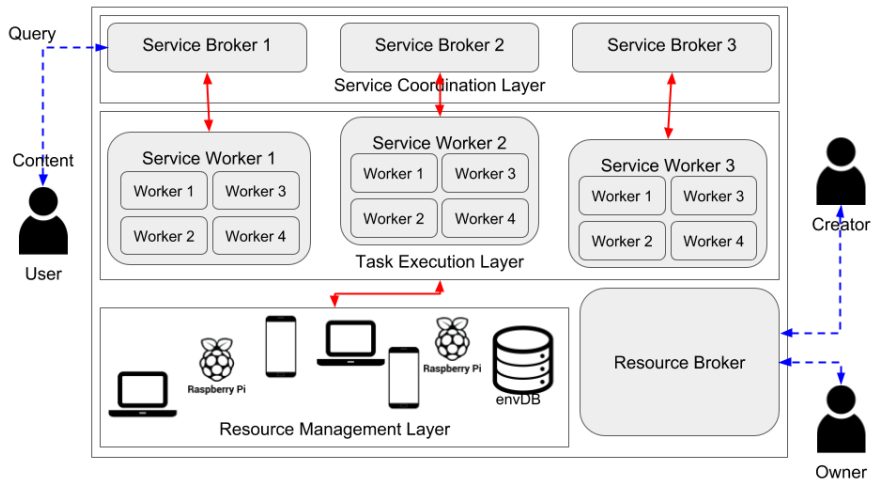

Figure 4.2: Improved IFoT Framework Architecture

edge devices. This is tackled by Challenge 3 in Chapter 5.

Security and Privacy: Security is one of the key challenges for IoT and IFoT [10]. It is important to ensure end-user/device authentication as well as maintain controllable data transmission between devices and IoT middleware. Data must be protected to ensure that data is not tampered with or leaked to adversaries. Also, user privacy is a critical issue since a user's habits, location, etc. can be collected without the user's awareness.

Device Discovery: IoT environments are usually heterogeneous and dynamic. Devices can change their availability and location at any time. Therefore, the middleware should support the discovery of new devices and/or data sources.

4.4 Improving the IFoT Middleware

In an effort to address the requirements stated above, improvements must be done on the IFoT middleware. **Architecture** and **Device Discovery** were tackled in the previous chapter and is a core part of the broker's functionalities. **Data Processing Support** will be discussed in the next chapter.

In order to deploy resilient and timely smart city services, the remaining requirements of **Latency** and **Security** must be tackled. We faced the following challenges for these requirements: (1) Node layout and network should prioritize

transmission delay and data security. (2) Services should be able to deal with data falsification attacks. (3) There should be a method to verify the efficiency of such configurations.

We solve these problems by improving upon the prior middleware we created. For (1), We measure the network and transmission delay between different configurations of brokers and workers. For (2), We implement anomaly detection over the service that deals with data falsification attacks. Finally for (3), we build a testbed that allows us to verify the effects of these on the service.

We present the design of an improved middleware meant for smart transportation systems with the added anomaly detection and task allocation algorithm. We introduce the testbed we created to evaluate these and show that our middleware performs better than the "naive" cases where resiliency is not performed and nodes are not configured in the optimal pattern.

To satisfy the requirements of the IFoT framework, we are developing a middleware platform [11] which allows services to be created by Service Creators. These services utilize the spatio-temporal data generated by sensors and processes it into useful information. The middleware system is comprised of a Resource Broker and Service Brokers described previously. Here we improve upon service workers.

Service Worker (SW): Service Workers, seen in Figure 4.2, are clusters of nodes that are able to perform operations on sensor data, and handle the computational tasks required to provide services to users. Each node executes tasks locally adhering to the shared-nothing architecture. To meet certain quality of service agreements, more nodes can be added to the cluster in order to scale up performance.

These are supported by three mechanisms: Environmental Database, Messaging Protocols and Task Graphs.

Environmental Database: stores the data generated by sensors. These are time series DBs stored in the SWs.

Task Graphs: are recipes that dictate how services are distributed and handled by the SWs. These contain instructions on how the SWs should collect, process and aggregate the sensor data for a particular service. These are generated by the SB taking into account service level agreements and QoS requirements to maximize the use of available nodes.

Publish-Subscribe-based Messaging Protocol (MQTT): used to facilitate communication between devices. Task graphs, heartbeat monitoring, and data for aggregation are sent via MQTT to the participating nodes (such as between SWs and SBs, or between SBs and RBs).

4.5 Resilient Smart Mobility Service

In this section, we present a service that takes advantage of the IFoT middleware platform. Middleware are often flexible enough to provide additional data that can also help in intrusion detection. These metadata can be passed alongside regular information, at the cost of some network overhead. This cannot be quantified well without an actual network in place. Thus, an emulation testbed is necessary to study the effects of such security measures.

Such a testbed can potentially be used to study the effects of different attacks (e.g. DoS attacks, botnet infection, etc.) on the network and its QoS. We investigate a novel decentralized anomaly detection approach for time-sensitive distributed smart transportation systems with a focus on data-integrity attacks and implement it in a testbed. Our method is based on related work in power systems [63], in which we extend in two distinctive ways. First, we extend the work for time sensitive applications. Secondly, we implement this approach in a decentralized network architecture.

The testbed would also need to be built into an IoT middleware platform which is capable of handling potentially large amounts of data in near real-time. Said platform would also need to be edge-based to minimize data transfer time. Combining an edge-based platform with such a testbed would be a novel approach in this case.

The end goal is to be able to test proposed security countermeasures for the issues detailed above. A smart traffic routing service is deployed on the platform to test its basic distributed processing capabilities, and then data is obtained from the integrated emulation testbed to quantify the overall delay introduced to the system.

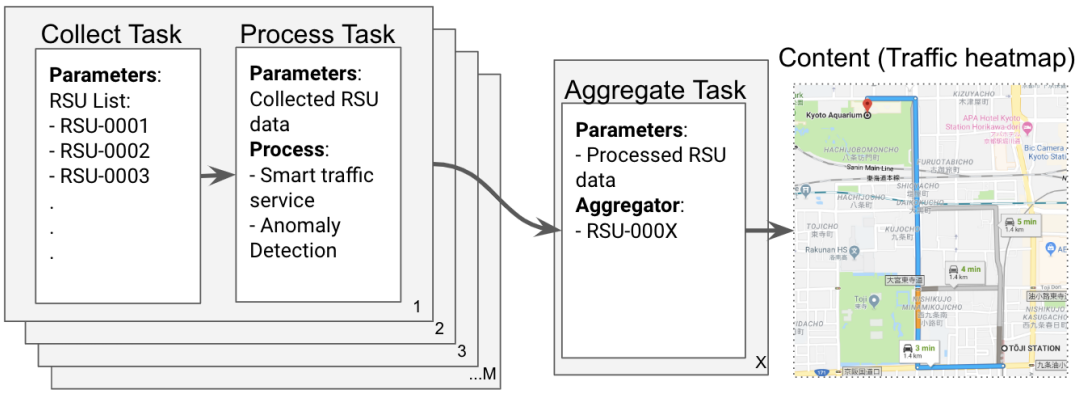

Figure 4.3: Service Broker generated Task Graph for M clusters

4.6 Assumptions of the Service

We assume that smart cities will feature roads and highways equipped with road-side units (RSUs). These RSUs receive information from vehicles such as speed. They are assumed to be devices with computational resources equal to those of Raspberry Pis or similar, and have their own wired network infrastructure allowing communication with each other.

RSUs connected to the IFoT middleware will host a smart mobility service that utilizes gathered data and publishes information about accidents, hazards, detours to its users. The service will also be able to respond to queries from users regarding the best routes for travel given the current situation.

Due to the spatio-temporal data being collected and processed by this service, it takes advantage of the properties of the IFoT framework for distributed computing. The middleware also allows security measures to be easily implemented within the service.

Other use cases While the service described above focuses on smart mobility, the infrastructure provided by the middleware can take advantage of any spatio-temporally distributed data. Distributed processing in this case decreases latency leading to improved QoS. The middleware also provides methods for introducing new services using various task graphs.

4.6.1 Details of the Task Graph

Task graphs dictate how services handle a user query, and are generated by the SB based on a particular service. Fig. 4.3 shows the task graph for the smart mobility service.

For this service, the task graph selects which RSUs will participate in the processing of the query. Selected RSUs would vary depending on the selected routes, desired QoS, delay requirements and the current load of RSUs. Afterwards, the tasks are distributed and executed. The task graph executes three different tasks.

- Collecting Task: The collecting RSU will query vehicular traffic data from other RSUs specified in the task's parameters. Once done, collected data will be distributed to other RSUs for processing.
- Processing Task: This includes all distributed processing that must be done on the data to produce the required result. In the case of smart mobility, traffic data will be checked for anomalies and then processed to generate route contexts (e.g. average speed information over a time-window, etc.) and other information for the user.
- Aggregation Task: Once all RSUs have finished processing, their results will be aggregated by one RSU and returned to the SB for visualization.

4.6.2 Resiliency

In terms of resiliency, we are primarily concerned with falsified data from orchestrated data-integrity attacks and hardware faults at RSUs and sensors. We define such attacks as scenarios where an attacker can compromise a subset of sensors or RSUs by manipulating sensor readings. At each RSU, an anomaly detection check is run at a specified time window using a statistical means detection method based on Bhattacharjee et al.'s approach to data falsification in power grids [63]. This work extends their approach in two distinctive ways.

First, we extend their work to time-sensitive applications. In the case of transportation, such an attack can have cascading effects on traffic behavior throughout the system. Since these effects are rapid, the time between the start of the attack and detection is critical. We address this by using anomaly detection time

windows ranging from 15 to 30 minutes. This method uses historical data to estimate the ratio of the harmonic mean to the arithmetic mean, which we refer to as the Q ratio. It was found on average that this process takes between one to two time windows to detect an attack.

Secondly, their work focuses on large-scale data. We extend this analysis to distributed transportation networks in which anomaly detection is run independently at each RSU and applied in the IFoT middleware. Using the testbed, we are able to quickly simulate and investigate the effects of various attacks on smart transportation networks, as well as the effects of the anomaly detection on network performance.

Hardware faults at sensors result in missing data readings at each time window, effectively simulating a large deductive attack. Thus, this model inherently extends to system failures in addition to data-integrity attacks.

As shown in Fig. 4.4, this metric proves to be stable over time, and responds quickly to simulated data-integrity attacks. This stability allows for easy integration with efficient sequential on-line anomaly detection methods as well as historical threshold methods. As anomaly detection done in this way does not depend on the other RSUs in the network, the detection process can be distributed throughout the network.

This approach provides numerous benefits over traditional anomaly detection, including simplified deployment over decentralized IoT networks. Additionally, the statistical means approach is computationally efficient compared to traditional anomaly detection methods such as Support Vector Machines (SVM), Bayesian Models, Gaussian Processes and Neural networks which require large scale, accurate models of system behavior and significant processing power. Statistical methods such as this have shown to be a more computationally efficient alternative [63]. This is particularly important for deployment on resource-constrained IFoT devices.

4.6.3 RSU Location Considerations

An important component of designing a smart city IFoT framework is in the placement of RSUs throughout the transportation network. In this sense, the number of RSUs available is a resource constraint in designing smart transportation grids.

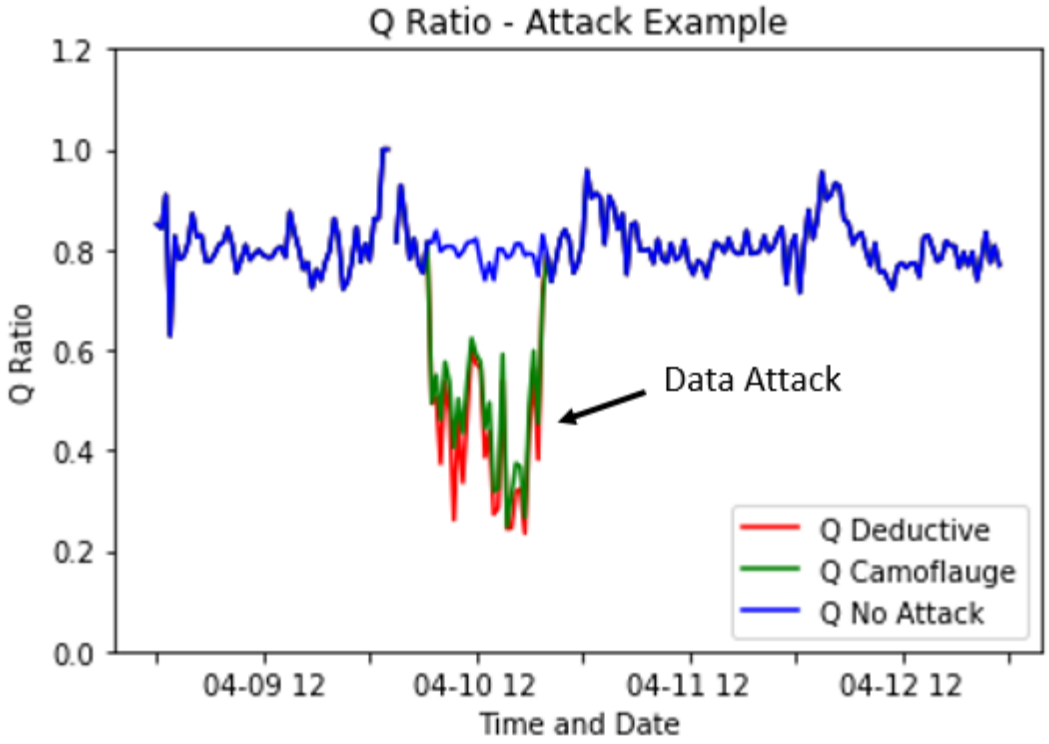

Figure 4.4: Q Response to Attack

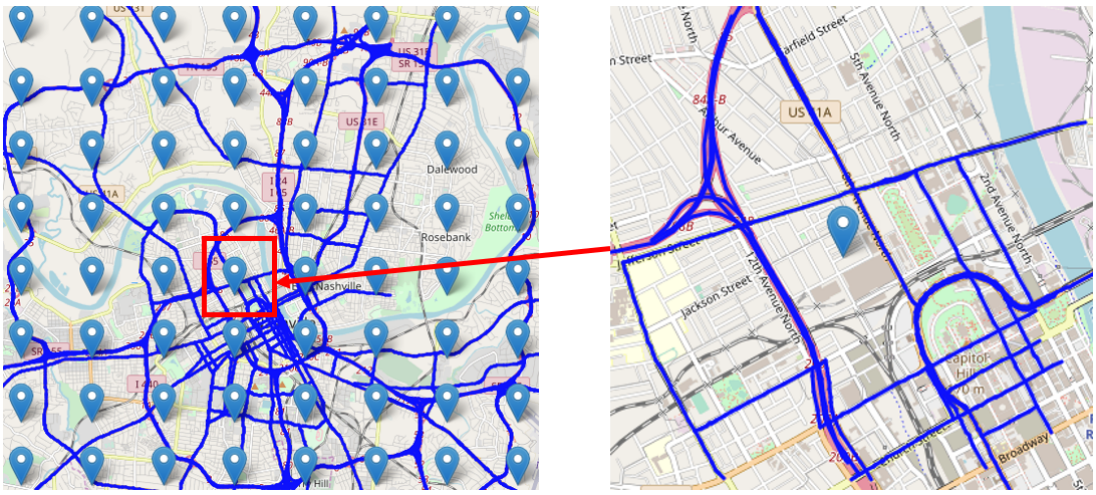

Figure 4.5: RSU Locations - Grid Layout

The major challenge is therefore determining the optimal spatial layout of these devices.

Optimal can be defined in any number of ways depending on device constraints and system goals. By designing a distributed testbed, optimal parameters such as delay and security can be compared between various system configurations through simulation. This reduces network design time and provides analytic feedback regarding expected system performance.

In the context of the transportation example, we focus on RSU layout in terms of network transmission delay and data security. We consider the case where each RSU is responsible for a subset of sensors streaming speed data into it. Thus, the RSU location problem is how to efficiently map these sensors to RSUs. The network layout is constrained by the number of RSUs available and the processing power of each device, corresponding to the number of sensors it can feasibly handle.

As a baseline RSU layout configuration we divide the city into 8×8 grids, resulting in 64 RSUs as shown in Fig. 4.5. A detailed investigation of delay performance for this configuration is provided in a later section.

Optimal RSU configurations can also be framed in terms of data-integrity resiliency. The effectiveness of the anomaly detection discussed in Section 4.6.2 increases for RSU zones consisting of sensors with traffic patterns similar and

dependent on each other. The grid layout provides a good proxy for grouping dependent sensors together. We look to improve this by providing a constrained hierarchical clustering approach in which sensors are grouped together by historical traffic pattern. To maintain network performance for data transmission between sensors and RSUs, we constrain the clustering procedure geo-spatially by restricting the maximum distance sensors can be from an RSU. Additionally we set a maximum number of sensors allowed per RSU in proportion to RSU processing capability.

4.7 Implementation

In this section we discuss how the platform is implemented on a testbed and how a service is deployed. To realize a testbed based on this architecture, it must meet the following requirements: (1) should be easy to deploy on heterogeneous IoT devices and should able to deal with heterogeneous data streams, (2) should have an area-by-area aggregation mechanism for spatio-temporal data streams, and (3) should be able to provide results in a timely manner, taking into account communication and processing delay between devices.

4.7.1 Testbed Implementation

As the platform should be easy to deploy on heterogeneous devices, it was initially implemented on Raspberry Pis with Debian using Docker for ease-of-deployment.

We developed a testbed to implement and test the middleware using various configurations. The platform could be deployed on the testbed to mimic a large number of nodes, simulated on a single 2018 Mac mini with 6-core 3.0 GHz i5 processor and 64GB of RAM. Each *SB* and *SW* is virtualized as a Docker service. To simulate constrained computation resources like Raspberry Pis, each service is assigned a limited amount of CPU and memory via Docker.

4.7.2 Service Simulation

Each RSU is assigned to a node and given unique parameters to simulate real world deployment scenarios. Each one is given location (latitude and longitude)

information as well as a unique ID for communication between RSUs. Each one also has an envDB that contains data received from the vehicles travelling along the road.

In our simulation, we divide a 80km^2 map of Nashville, TN into 8×8 grids, where each vertex corresponds to an RSU collecting data from vehicles in specific sections of the road network seen in Fig. 4.5. To evaluate the system, each RSU is set to behave as either a *SW* or normal Worker. RSUs are grouped into clusters with a single *SW* and one or more Worker nodes. Data for the simulation uses the 2014 Nashville city roawd records [64] which collected speed data from vehicles travelling the roads. These roads contain sensors placed at specific traffic message channel (TMC) points which make up a segment of a road. A combination of speed data, TMC points and optimal RSU locations are used to determine which RSU will store which road's data in their envDB.

Connections between RSUs are assumed to use wired Ethernet connection. In order to simulate the real world work flow of this service, users are able to query the platform through the Service Broker's web interface.

It is assumed that the SB determines the route the system will select in response to the query of the user. The system represents the variations of these routes as the variations of the number of clusters and workers within the cluster. Once the user has successfully sent a query to the SB, the execution timer starts and the task graph for the service is sent to these clusters.

4.7.3 Delay Emulation

Since the RSUs are simulated, data transfer delays are obtained via synthetic calculations and injected wherever such communications happen. For instance, after the collection task, the RSU handling it passes off results to other RSUs for processing. A short delay is injected here through a sleep function right before data is sent to each processing RSU. Since the simulated RSUs are also doing real processing on the data, delays due to processing will be left as is.

$$d_{SB \to SW} = d_{GW} + d_{CL} \tag{4.1}$$

Execution time measurement starts after the user query arrives at the *SB*. The

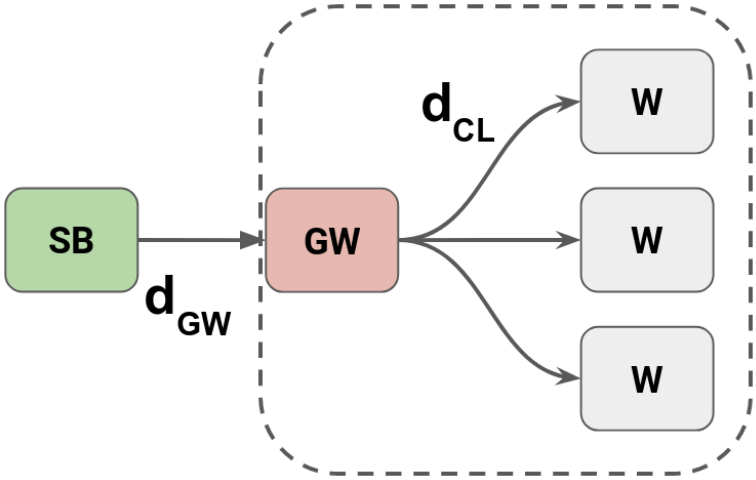

Figure 4.6: Injection Points for Delay Components in Simplified System

SB publishes a message to the cluster gateway which then routes that message to the *SW*. This introduces a delay given by Eq. 4.1, where d_{GW} is the delay between the *SB* and the cluster gateway and d_{CL} is the delay between the cluster gateway and any one of the cluster's RSUs.

$$d_{W \to W} = 2d_{CL} \tag{4.2}$$

The *SW* then performs the task designated for it on the received task graph. Usually, it is given the collection task and the result - along with the task graph - is passed on to other RSUs. These then proceed to perform their designated tasks, using the results passed from the previous step. Passing data between RSUs introduces a delay as shown in Eq. 4.2.

$$d_{W \to SB} = d_{GW} + d_{CL} \tag{4.3}$$

Once all RSUs have finished processing, resulting data must be aggregated back at the *SB*. The delay for this is given by Eq. 4.3. Fig. 4.6 summarizes where these delay components are injected in a simplified system.

$$d_{comp} = d_{trans} + d_{prop} \tag{4.4}$$

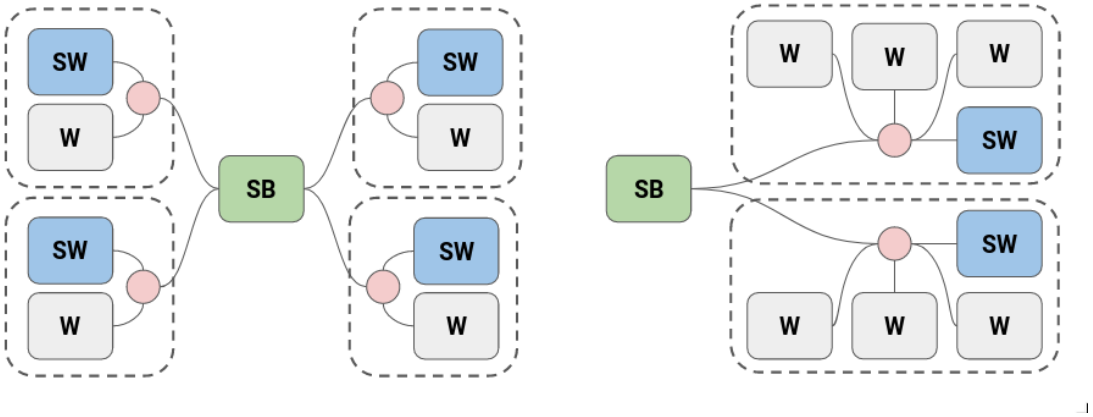

Figure 4.7: Different possible network configurations for a system with 8 nodes. (Left) 4 clusters with 2 workers each, (Right) 2 clusters, with 4 workers each

Each delay component is broken down further into relevant parameters as shown in Eq. 4.4, where they are defined as:

$$\begin{aligned} d_{trans} &= data\ length / bit\ rate \\ d_{prop} &= link\ length / propagation\ rate \end{aligned} \tag{4.5}$$

The *link length*, *bit rate*, and *propagation rate* can be configured for the simulation, while *data length* is based on the actual quantity of data sent during run-time. For this simulation, the values used are shown in Eqs. 4.6 and 4.7.

$$d_{GW}: \begin{array}{l} bit\ rate = 2Mbps \\ link\ length = 17.6km \\ propagation\ rate = 299.792 * 10^6 m/sec \end{array} \tag{4.6}$$

$$d_{CL}: \begin{array}{l} bit\ rate = 2Mbps \\ link\ length = 8.0km \\ propagation\ rate = 299.792 * 10^6 m/sec \end{array} \tag{4.7}$$

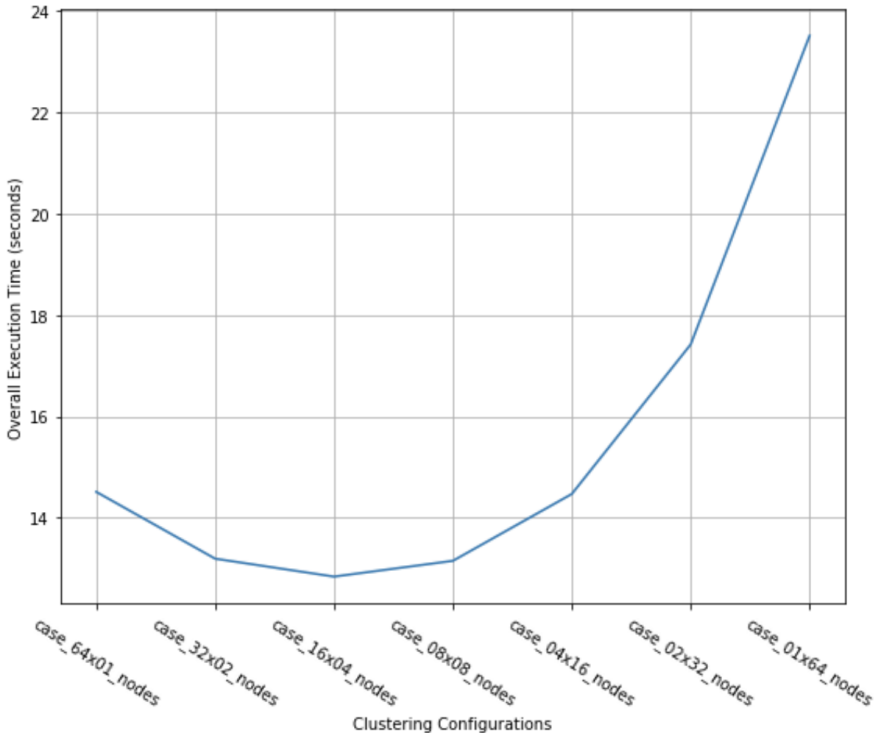

Figure 4.8: Overall Execution Time vs Clustering Configurations

4.8 Evaluation

In order to evaluate the system's utility, we use it to measure the overall execution time of a service implemented over IFoT. Overall execution times of tasks processed on a distributed network can vary depending on the network architecture. In this simulation, the network can be configured in multiple ways as shown in Fig. 4.7. For example, with 8 total RSUs, Fig. 4.7a shows how we can split RSUs into 4 clusters, each having 1 RSU as the *SW* and 1 dedicated Worker. Fig. 4.7b shows how the same network can be configured as 2 clusters, each with 1 *SW*, and 3 dedicated Workers.

For example, the testbed can be used to identify which configuration will lead to the least overall execution time. Using a Command Line Interface (CLI), this experiment was repeated with varying combinations of clusters and workers within a cluster. Fig. 4.8 shows the total time for the different combinations of clusters and workers. X axis represents the different combinations with the

following naming convention: *case_AAxBB_nodes* where *AA* is the number of workers in a cluster and *BB* is the number of clusters in the system.

In this case, the best clustering configuration (one with least processing delay) is found at **case_16x04_nodes**, or where there are 16 workers in each of the 4 clusters. When we attempt to increase the clusters and instead decrease the number of workers, the delay increases by more than double. **case_01x64_nodes**, with 64 nodes in a single cluster, experiences the most delay. This delay only decreases as more nodes get included in a single cluster. On the other end of the spectrum, when you have a single cluster with all 64 nodes, the delay increases but only by a negligible amount. This result show that the communication delay between clusters are the main factors which cause the bottleneck in execution time. This may be due to the scheduling and task allocation that must be done with more clusters available.

In order to verify this further, we have to test this with a control group, where the number of clusters stay constant and vary the workers or vice versa. However, from this small sample size, we can still see that the testbed allows us to draw preliminary conclusions about how changing the configuration of the simulated network affects execution speed. However, this result does not apply to all middleware. The effect of the clustering is only valid for this specific scenario and set of parameters. The goal of the chapter was to create a tool that would let us know in advanced, the delays that will be caused by varying parameters and clusters.

4.9 Summary

In summary we designed and developed a middleware platform that meets IFoT framework requirements. This allows for high availability and low latency communication. The middleware is suited for services that deal with spatio-temporal data. In addition, we identify configurations of the middleware which ensure resilient and timely services over the framework.

We show this by developing a testbed that is highly configurable and easy to setup. A smart transportation service was developed and deployed on the testbed to demonstrate how the middleware deals with spatio-temporal data on multiple

nodes. As an example, we show that the testbed can be used to analyze how the middleware can meet certain QoS level requirements by configuring its network architecture. Experimental security measures on top of the middleware could be evaluated in a similar way.

5 Route Planning through Distributed Computing using Road Side Units as Resource

5.1 Introduction

In this chapter, we extend our work by considering route planning as another smart city service that can be implemented using the IFoT middleware presented in Chapter 3 and 4. Here, we face the problem of realizing a route planning service that uses data from distributed road side units (RSUs) in order to generate its routes.

Cities are already investing in deploying RSU networks in part for the preparation of autonomous and connected vehicles. These devices which are low powered Raspberry Pi-like devices placed all along city roads and highways. Since each RSU is capable of limited amount of processing and storage, when multiple RSUs are connected in a sub-network, they form a fog computer [6] and can be used for route planning services. This sub-network is a private, reliable and pervasive network that cities and its citizens can use. Prior work has been done showing that IoT devices can be used in routing for emergency management scenarios [65] and while also allowing vehicle-to-vehicle communication for navigation [66].

While route planning is a well-studied topic, state-of-the-art route planning algorithms [46], [47], [48] are developed typically as centralized approaches for centralized architectures such as cloud environments and data-servers. In these scenarios, data is shared and parallelized, allowing for typical search algorithms to access to a shared memory and direct communication between multiple processors. These algorithms are not suited for a distributed setting.

The major challenge is in designing route planning algorithms that work well in a distributed setting such as with RSUs. The idea behind our approach is to divide the city into grids which we then assign RSUs to. Each RSU has its own local data for the grid it covers as well as models for route planning.

In this chapter we show that routing algorithms designed specifically for fog computing such as for the RSUs at the edge, require more attention to task allocation given the processing and memory constraints of each device. Due to the distributed nature of the system, data is not stored in a central location so there will be trade-offs between processing delay and accuracy. Our approach has significant speed gain compared to centralized approaches and offer system reliability and data security.

The rest of this chapter shows our assumptions on the service, the components and the workflow of the system, the distributed task allocation problem and finally our solutions and evaluations.

5.2 System Architecture

This section describes assumptions on the target regional area, decentralized route planning service, and the tasks generated by user queries. Table 5.1 summarizes the symbols used throughout this section.

Table 5.1: List of symbols

Symbol	Description
G_r	Grids making up the target area
RSU_i	Road Side Unit i
R	Road Side Unit assigned to a grid
N	Network graph, $N = (V, E)$
V	Road intersections
E	Set of roads
d_i	Local speed data generated by sensors in a grid
Q_p	Set of queries sent within in the same time period p
G^i	Subgraph whose edge and vertex maps to grid i
Query	
q	Query sent by the user
s	User source location
d	User destination location
τ	User desired travel time
Route Planning	
SG_q	Sequence of grids (s to d) generated per query q
k_q	Multiplier based on q parameters
T_q	Sequence of tasks $t_{q,i}$ for each SG_q
$t_{q,i}$	Route planning tasks assigned to each grid in SG
Task Allocation	
T	Set of all tasks T_q for all queries Q_p
$x_{t,r}$	Assignment variable, 1 if task is assigned, 0 otherwise
$ST(t)$, $ET(t)$	Task execution start and end times
D_{th}	Delay threshold for query responses
$TQ(th)$	Threshold for assigned queries per RSU
$IT(q)$	Time when q was issued
$ct(t, r)$	Computation time of task t executed at RSU r
MA	Model accuracy of the route planning model
$MD(g, g')$	Manhattan Distance between two different grids

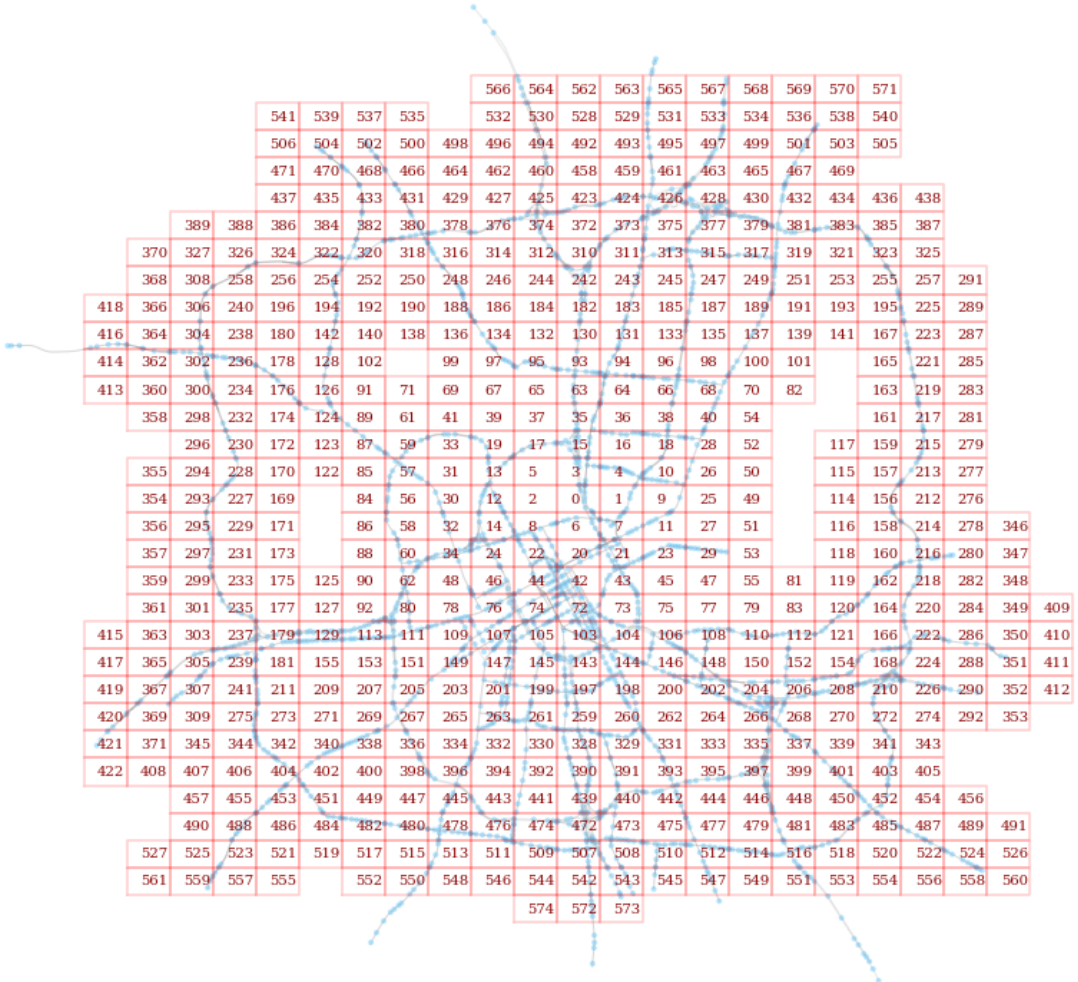

Figure 5.1: Target spatial area is divided into grids

5.2.1 Spatial Region

We assume that smart cities are equipped with sensors and computational resources that allow it to monitor and optimize their resources to maximize its services to its citizens. We assume that traffic lights and lamp posts, equipped with road side units, exist along the city roads and highways.

The target area is split into equidistant *grids*, as shown in Fig. 5.1, which we denote as $G_r = \{g_1, g_2, \cdots, g_m\}$. *RSUs* are then deployed on these grids, the typical distribution is one RSU per one grid. Each RSU is connected to a *regional area network*[*].

Each RSU is assumed to have storage and computational resources to store sensor data as well as execute tasks from a task queue. The distributed network is denoted as a resource graph with a set of vertices $R = \{r_1, r_2, \cdots, r_m\}$. Each vertex represents an RSU over the subarea, while each edge is an undirected link between any two RSUs. The physical area map is represented by a network graph $N = (V, E)$ where V are road intersections and E are road segments.

RSUs are assumed to be both resource and memory constrained (e.g., Raspberry Pi-like computation power). As such, each RSU can only accommodate a finite amount of simultaneous tasks and hold a finite amount of sensor data and machine learning models. We assume that mobility data, generated by passing vehicles, are sent to sensors nearby or within the RSUs.

5.2.2 Decentralized Route Planning Service

This service runs on the middleware that utilizes the distributed characteristics of the architecture. This system allows users access to time-dependent and privacy-preserved services for smart transportation. With respect to the problem of consistency, availability and partitioning for distributed systems, the middleware is able to work under partitioning. Since RSUs, each with their local data, form their own sub-network, the service is able to maintain availability even with the loss of cloud services. Services that make use of geospatial data, such as smart grids and smart mobility, can effectively use this system. However, for this dissertation, we focus on implementing a *decentralized route planning service*.

[*]We assume that RSUs are connected with wired links.

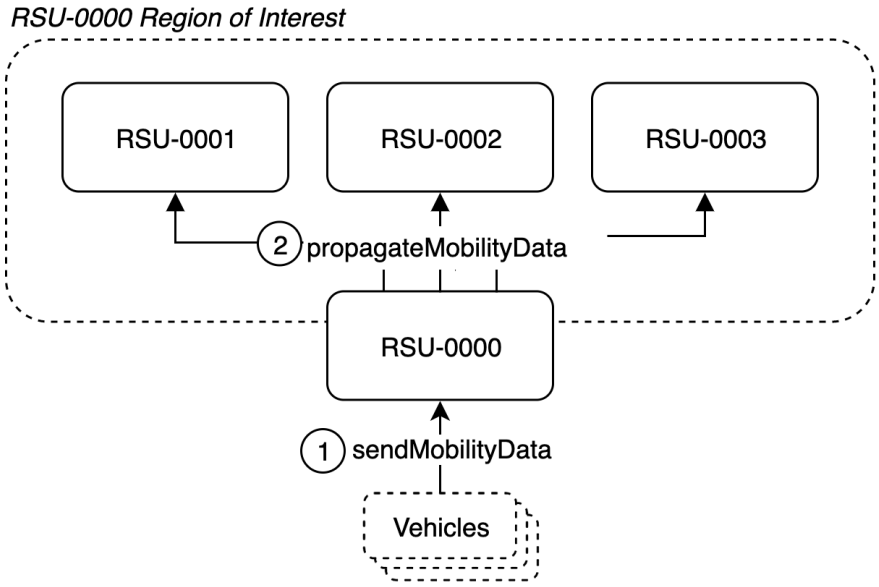

Figure 5.2: Data gathering and propagation architecture of the middleware. Each RSU gathers mobility data from vehicles in their area and then propagates them to other RSUs within their search area.

We assume that each grid contains several road segments and each road is assigned an RSU that handles all incoming speed data.[†]. Each RSU has its own local storage for any static data that can be loaded before hand or stored after gathering. Each RSU $r_i \in R$ receives mobility data from vehicles on roads within their designated grids g_i.

Each RSU stores up-to-date speed data only for the roads of the grid it is assigned to. Each RSU propagates its aggregated data to neighboring RSUs at a set delay interval. Data shared with neighbors is assumed to not be as up-to-date as data found at the data source. This historical data distribution alleviates communication costs between RSUs while allowing the system to maintain a snapshot of the whole target area's speed information for route planning.

Our system consists of the following types of data:

1. *Mobility data*: Periodically collected up-to-date speed data from vehicles that pass along roads within the RSU sensor's range. These are stored as

[†]These RSUs are all connected but have one RSU as the representative of the grid, which we refer to in this section.

time-series data with both road speed and location information. These are used as the weights of the network data.

2. *Historical data*: These are assumed to be data that has been averaged and propagated to neighboring RSUs within a region of interest. This is not up-to-date data and are snapshots of neighboring RSUs data at a previous time frame.

3. *Network data*: Each RSU maintains a list of subgraphs, G^i, extracted from the global routing graph $N = (V, E)$.

For this service we follow a local storage architecture. The three types of data stored in each RSU is inherently localized. This is because mobility data is highly localized in each grid's coverage area. Data from vehicles traveling in each road within the RSU's coverage are received by each RSU and stored as time series data. These data are then aggregated into historical data and distributed to its neighbors.

Figure 5.2 shows the methods used to propagate data between RSUs. Passing vehicles call ① `sendMobilityData` to the nearest RSU on the road. At some time interval or delay, each RSU will call ② `propagateMobilityData` to all the RSUs within its region of interest. All RSUs within the target area perform this.

Cloud storage is not utilized, since we assume that data collecting, aggregation and processing is done only on edge devices with minimal to no assistance from any cloud-based services or servers.

Finally, the network data held by each RSU is static and contains only the road network information, roads and intersections, of the particular area within the RSUs coverage. This data does not change and is not particularly large. This is true regardless of how large the grid area is. Given these, it is easy to load preemptively and distribute to other RSUs when needed.

5.2.3 User and Query Tasks

Each RSU $r \in R$ is able to receive some query q with parameters (id, s, d, τ_s). id is used to differentiate the queries while s and d are the route's desired start and destination points respectively, and τ_s is the user's desired departure time.

Queries can be received asynchronously by multiple RSUs at some time window i, we denote these sets of queries as $Q_i = \{q_{i,1}, q_{i,2}, \cdots, q_{i,n_i}\}$, where i is the time window the particular set of queries were received.

For each received query, we assume that a corresponding sequence of grids where optimal travel path is likely included will be generated[‡]. We call this $SG_q = \langle g_{q,1}, g_{q,2}, \cdots, g_{q,k_q} \rangle$ where q is the query while k_q is a modifier that is dynamically generated and is based on the query as well. This sequence of grids corresponds to a grid level view of the route the application serves the user.

For every SG_q, a sequence of tasks $T_q = \langle t_{q,1}, t_{q,2}, \cdots, t_{q,k_q} \rangle$ where $t_{q,i}$ is the route planning task in grid $g_{q,i}$ is also generated. These tasks include getting the travel time from one vertex $v \in V$ via a road segment $e \in E$ as well as creating the path through these using path search algorithms such as Dijkstra's.

5.3 Distributed Route Planning

The distributed route planning service needs to provide a *shortest route* from a source to a destination for a given departure time. User's queries can be sent to any RSU which then serves as the broker for this particular query. The architecture is seen in Fig. 5.3, describes all major components of the platform. Each edge includes a circled number, i.e, ⊛.

5.3.1 Definition of the Problem

For a target area that is populated by a set of RSUs R, a set of user queries q is sent within a period p. For each q, an optimal route is generated by ② `getOptimalSequenceGrid`. The route is broken down to its corresponding grids and divided into separate tasks by ③ `generateTasks`. The set of all tasks T for all queries Q must be allocated to RSUs R for processing. The problem thus, identifying the most efficient and optimal allocation of tasks T to RSUs R. All

[‡]We assume that each RSU has a model to compute the next grid toward the destination point with inputs of the current grid, the source and destination points and start time. By repeating this next grid computation, we can get the sequence of grids from the source to the destination point.

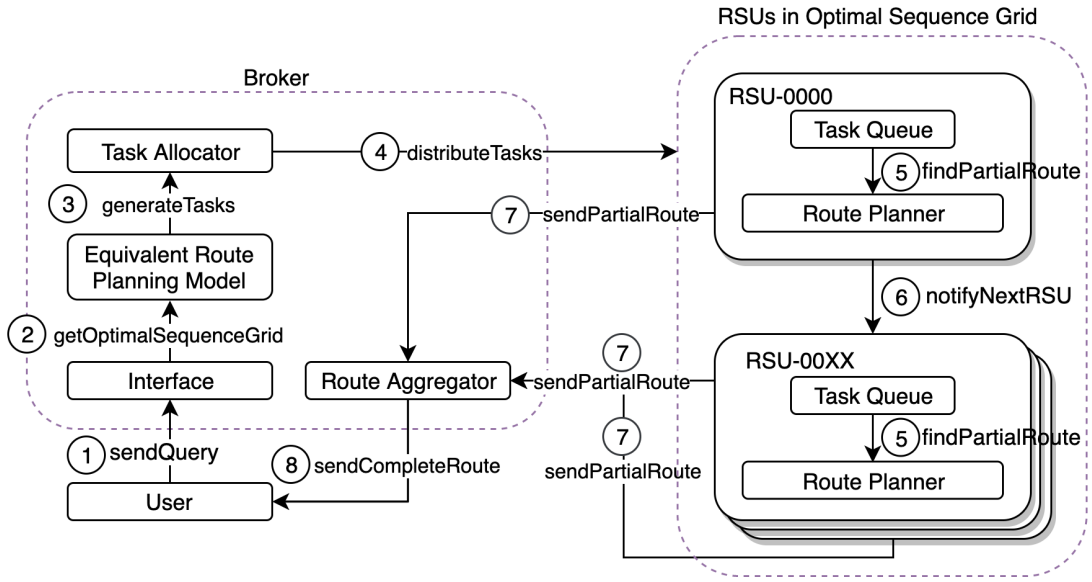

Figure 5.3: Distributed Route Planning architecture of the middleware and the flow of interaction between components. Once the user sends a query to the Broker, `getOptimalSequenceGrid` generates an optimal route for the query. The query is then divided into tasks which are then sent to corresponding RSUs by `distributeTasks`. Each `findPartialRoute` and `sendPartialRoute` is run in a sequential manner. Once all routes have been sent to the broker and aggregated, the user receives the complete route.

task assignments should satisfy the constraints on query response delay while maximizing the overall accuracy of the result.

Delay

We set Eq. 5.1 as the first constraint. We define T as the set of all tasks T_q of all queries $q \in Q_p$, sent at time period p.

$$T \triangleq \bigcup_{q \in Q_i} T_q \tag{5.1}$$

For every pair of task $t \in T$ and RSU $r \in R$, we define a variable $x_{t,r}$ which is

1 if t is assigned to r and 0 otherwise.

We assume that every task $t \in T$ is assigned to a single RSU, so the following condition must hold.

$$\forall t \in T, \sum_{r \in R} x_{t,r} = 1 \tag{5.2}$$

In each sequence of tasks T_q for query q, tasks must be sequentially executed. Hence the following equation must hold. Here, $ST(t)$ and $ET(t)$ represent task execution start and end times, respectively. Here $t_{q,i}$ is the current task being executed at grid i and $t_{q,i+1}$ is the next task in the sequence T_q, to be executed in the next grid $i+1$ in sequence. Here k_q is the last grid in the sequence for query q.

$$\forall T_q (q \in Q_i) \forall i (1 \leq i \leq k_q - 1)\, ET(t_{q,i}) < ST(t_{q,i+1}) \tag{5.3}$$

Upon assignment of all tasks in T, we define that the overall service delay for queries Q_i, should not exceed some delay threshold D_{th}. Here, $IT(q)$ is the time when the query is issued.

$$\forall q \in Q_i, ET(t_{k_q}) - IT(q) \leq D_{th} \tag{5.4}$$

Also, we define that the number of tasks t that can be queued on any single RSU r should not exceed the queue length threshold TQ_{th}. Here $TQ(r)$ is the number of tasks queued on RSU r.

$$\forall r \in R, TQ(r) \leq TQ_{th} \tag{5.5}$$

However, in cases where all RSUs have reached the TQ_{th}, the task is allocated to the least utilized available neighbor RSU. The purpose of Eq. 5.5 is to facilitate task load distribution over all the RSUs.

Each RSU r must first execute all tasks, within its task queue, ahead of task $t_{q,i}$. The worst-case execution time of task $t_{q,i}$ will be the sum of worst-case execution times of all tasks ahead of it in the queue.

Then, task execution start and end times of each task $t_{q,i}$ can be defined as follows.

$$ST(t_{q,i}) \stackrel{def}{=} IT(q) + \sum_{j=1}^{i-1} \sum_{r \in R} \sum_{t' \in T} ct(t', r) \cdot x_{t',r} \cdot x_{t_{q,j},r} \tag{5.6}$$

$$ET(t_{q,i}) \stackrel{def}{=} ST(t_{q,i}) + \sum_{r \in R} ct(t_{q,i}, r) \cdot x_{t,r} \tag{5.7}$$

where $ct(t, r)$ represents the computation time of task t executed at RSU r. These are given in advance.

Equation 5.6 defines that task, $t_{q,i}$ can only be executed after all prior tasks from the same query, $(t_{q,1}, ..., t_{q,i-1})$, have been processed. This means that the result of these tasks, such as *travel time* and *shortest paths*, has been generated.

Accuracy

For every query, Q_i, a set of tasks T_q is generated based on the optimal sequence grid, $SG(q)$. Assigning tasks to the grids in SG produces the highest accuracy for route planning. However, due to the computational and memory constraints of the RSUs, tasks often take longer to be processed than to be assigned. As the task queue increases, the execution time of any additional tasks will increase as denoted by Equation 5.6. This increase in total execution time results in query response delays. A solution would be to allocate certain tasks onto less utilized but sub-optimal neighbor RSUs.

We assume that all RSUs only have up-to-date access to their local data and access to only stale data from neighboring ones. We assume that RSUs propagate their data to nodes within their region of interest, using gossip-based protocols [67]. The neighboring RSUs pass the received information, as well as their local data, to their neighbors. The size and rate of propagated data decreases the farther the receiving RSUs is from the data source. Thus tasks allocated to sub-optimal RSUs produce less up-to-date, but nonetheless correct route from s to d. The staleness of this data is given by the differences between the optimal and actual assignment locations as follows:

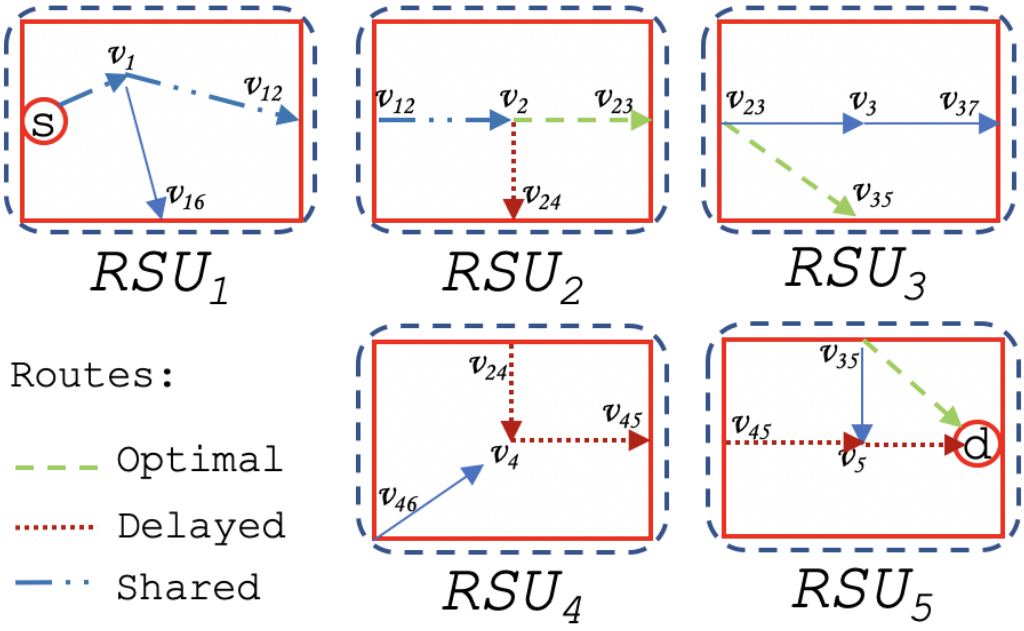

Figure 5.4: Decentralized Route Planning example to show the effects of accuracy and delay

$$MA(g, g') \triangleq 1 - MD(g, g') \tag{5.8}$$

where $MD(g, g')$ is a factor of the Manhattan Distance between the optimal and actual grid assignments, while $MA(g, g')$ is the estimated model accuracy difference due to task allocation.

Impact of Accuracy and Delay

To demonstrate the impact of delay and accuracy on the route planning output, we use a sample route, Fig. 5.4, with 5 RSUs and a trip query from source s to destination d. An optimal route will have all its tasks allocated to optimal RSUs. However, this optimal route might exceed the D_{th} shown in Eq.5.4. If all the tasks are forced to optimal RSUs, up-to-date data is used resulting in high accuracy at the cost of processing time.

When tasks are re-allocated to sub-optimal RSUs such that the new allocation sequence does not exceed D_{th}, the route from s to d is still obtained. However, the data used may not be up-to-date, given by Eq. 5.8, resulting in a route of lower quality. A low-quality result does not mean an error in the routing, instead it means that the data being used to generate the route is not up-to-date resulting in a different route (dashed vs dotted line) to the same s to d. This increases the chances of using congested routes. The trade-off is that these sub-optimal RSUs will be able to process the tasks faster since they meet the D_{th} and/or TQ_{th} constraints.

Utility Function

Given the example above, we assume users have two requirements from the service. First, to receive a response within a preferable time delay and secondly, to receive it with an acceptable degree of accuracy. Based on these two requirements, we design the utility function $U(q)$ as follows. At every time i, the tasks T_q generated by a query q, should be assigned to RSUs such that it meets the constraints while maximizing the accuracy of the generated route.

$$U(q) = \sum_{j=1}^{k_q} \frac{MA(g_{q,j}, g(t_{q,j}))}{k_q} \tag{5.9}$$

Objective Function

The purpose of distributed task allocation is to find the most optimal assignments of tasks to RSUs that satisfy the given constraints while maximizing the accuracy of the generated routes. The objective therefore is:

$$\textbf{Maximize:} \sum_{q \in Q_i} U(q) \text{ subject to } (5.2) - (5.5) \tag{5.10}$$

5.3.2 Region of Interest Heuristic

To meet the objective function, our system needs to divide queries into tasks and then allocate them to RSUs such that accuracy is maximized while constraints

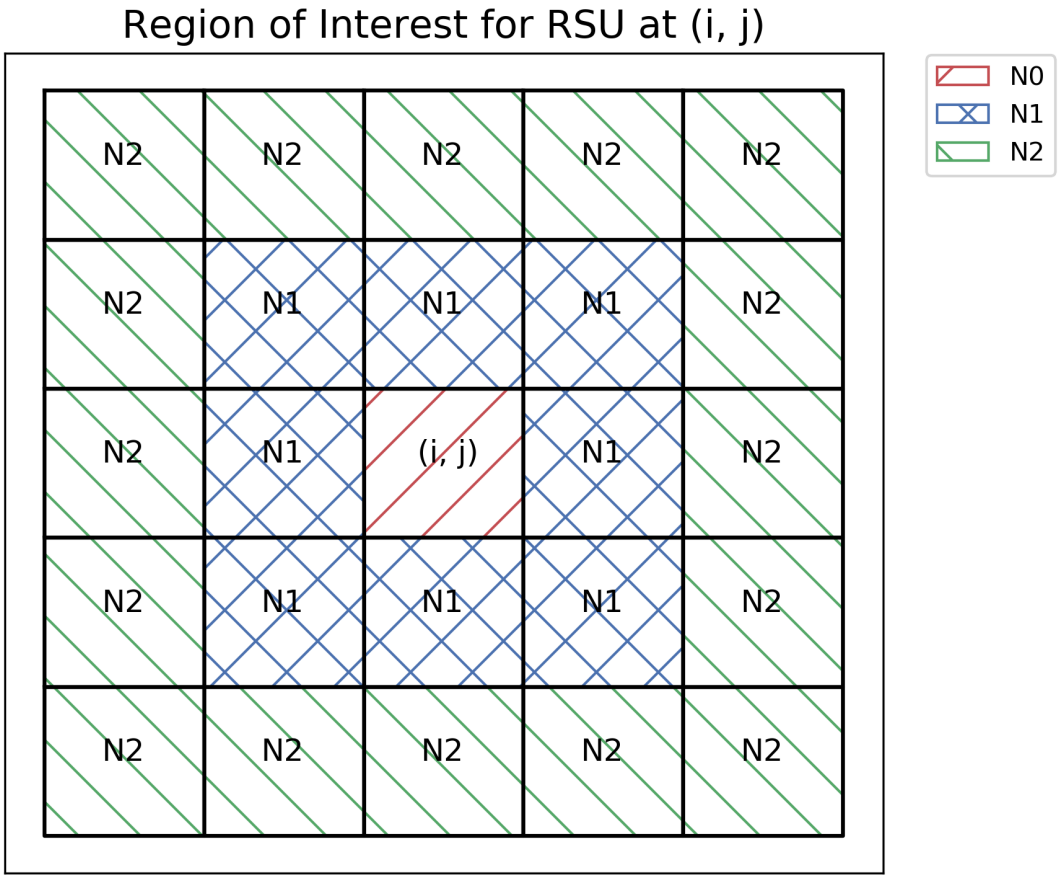

Figure 5.5: Region of interest based on varying Neighbor levels for RSU at the location (i, j). Neighbor level 1 includes the N0 grid at (i, j) and all N1 grids. Level 2 includes all grids N0, N1, and N2. As neighbor level increases, more grids become alternative grids.

Algorithm 1 `getOptimalSequenceGrid`

Input: Source $s \in V$, Destination $d \in V$, Time: τ

Output: Optimal sequence grid OG

Initialize SeqGrids list
$i \leftarrow 0$
$SeqGrids[i] \leftarrow$ GetGrid(s)
$g_{final} \leftarrow$ GetGrid(d)
while $SeqGrids[i] \neq g_{final}$ **do**
    $currentGrid \leftarrow$ SeqGrids[i]
    $SeqGrids$[i+1] $\leftarrow$
    GetNextGrid($currentGrid, g_{final}, \tau$)
    $i \leftarrow i + 1$
end while
SeqGrids[i] $\leftarrow g_{final}$
return SeqGrids

Algorithm 2 Get Next Grid

Input: Current Grid: g_{curr}, Destination: g_{dest}, Time: τ

Output: Next Grid: g_{next}

Get Equivalent Grid Routing model $\hat{E}$
$g_{next} \leftarrow \hat{E}$.predict($g_{curr}$, g_{dest}, τ)
return g_{next}

are met. Region of interest controls the search area for RSUs that can be used to handle tasks that cannot be allocated to optimal RSUs because of over-utilization. Over-utilization occurs when the RSU exceeds either D_{th} or TQ_{th}, such that further allocation to it will cause delay on the system.

For our approach, we vary the region of interest using the neighbor levels. A level of 0 allocates tasks only to the most optimal RSUs as decided by SG. This prioritizes accuracy over the processing time. Figure 5.5 shows how the levels affect the region of interest. For levels 1 and 2, we increase the region of interest to the surrounding 8 and 24 RSUs around the optimal RSU respectively. The wider the region of interest, the greater the number of RSUs available for task reallocation. However, the wider region of interest also affects the distance

Algorithm 3 Sequence Grid Task Allocator

Input: Set of queries: Q

Output: Modified Grid: MG

1: **for all** $q \in Q$ **do**
2: $OG_q \leftarrow$ getOptimalSequenceGrid$(q.s, q.d, q.\tau)$
3: **if** $Delay(OG_q) > D_{th}$ **then**
4: $OG_q \leftarrow$ ModGSeq(OG_q)
5: **end if**
6: distributeTasks(OG_q)
7: **end for**

Algorithm 4 Modified Sequence Grid Generation

Input: Sequence Grid: SG, Neighbor Level: L

Output: Modified Grid: MG

1: $MG \leftarrow SG$
2: **while** $Delay(MG) > D_{th}$ and $|SG| \neq \emptyset$ **do**
3: $g \leftarrow$ a grid randomly selected in SG
4: $ns \leftarrow$ GetGridNeighbors(g, L)
5: $bg \leftarrow$ GetLeastUtilized(ns)
6: $MG \leftarrow$ modified MG by replacing g with bg
7: $SG \leftarrow SG - \{g\}$
8: **end while**
9: **return** MG

between the optimal RSU and the selected RSU, given by Eq. 5.8.

5.3.3 Decentralized Route Planning Example

To demonstrate the overall execution of the route planning service, we use Fig. 5.4 which shows a network partitioned into 5 RSUs. This network is equipped with the middleware shown in Fig. 5.3. The information flow is initiated by a user calling ① `sendQuery` with parameters (id, s, d, τ_s) to RSU_1. The rest of the flow follows the numbered components in Fig. 5.3.

Upon receiving the query from the user, RSU_1 calls ②

`getOptimalSequenceGrid`, which executes Algo. 1. This uses the source, destination and time parameter of the query as well as the *Equivalent Grid Routing model* ($\hat{E}$) [68], to recursively identify the optimal grid sequence SG, from the source s to the destination d. $\hat{E}$ is a routing model used by Algo. 2, that predicts the best neighboring grid through which the shortest path likely resides for a particular route. $\hat{E}(s, d, \tau_s)$ returns the next best possible grid to travel to destination d from s at time τ_s.

Here, the optimal sequence grid generated is as follows: $[RSU_1, RSU_2, RSU_3, RSU_5]$. A set of tasks for this sequence is then generated by ③ `generateTasks`. These tasks are then run through the task allocator described by Algo. 3. Algorithm 3 first assigns tasks to the optimal sequence grid, and then it measures the total delay given this configuration. If such a configuration causes a delay greater than D_{th}, we modify the sequence grid using Algorithm 4.

In Algorithm 4, we pick the most utilized RSU in the optimal grid sequence SG, and then we select an RSU within its region of interest. We move this RSU's task assignment to the least utilized RSU in that region of interest, modifying the current sequence of grids in response. The total delay of this new sequence grid is again checked against D_{th}.

This process is repeated until the total response time for the query is less than D_{th}. We then assign the tasks to this sequence of grids, making sure to consider the task queue constraint in Eq. 5.5, during the allocation.

In this example, we assume that the modified sequence grid is now $[RSU_1, RSU_2, RSU_4, RSU_5]$. The two-fold impact on accuracy and delay of the allocation algorithm was discussed in Section 5.3.1. Once the task allocation algorithm has verified that constraints have been met or search has been exhausted, tasks are distributed to the RSUs following the modified sequence grid with a call to ④ `distributeTasks`.

Tasks are distributed in parallel and executed sequentially (Eq. 5.3). RSU_1 processes the task once it appears at the start of the queue. It generates a partial route with a call to ⑤ `findPartialRoute` and sends it to the broker for aggregation with ⑦ `sendPartialRoute`. At the same time, it notifies the next RSU in sequence, RSU_2, via ⑥ `notifyNextRSU` and the process continues. Once the

final RSU, RSU_5 sends its partial route, the broker calls ⑧ `sendCompleteRoute` to send the final route to the requesting user.

For tasks to be optimally executed, tasks must be performed sequentially on the optimal sequence of grids. The accuracy of the service depends on the distance between optimal allocation and actual allocation, thus Tasks are allocated by the algorithm which considers queue lengths and neighbor RSUs. This restriction prevents other selection mechanisms from working. Optimizing node selection using techniques such as integer linear programming (ILP) or multi-objective genetic algorithms (MOGA) may work when tasks can be allocated in any node or RSU within the target area.

5.4 Experiment and Results

In this section, we evaluate our distributed route planning service over the middleware. We evaluate the system in two phases. First, we evaluate the system without devices in the loop. This is done to prove the feasibility of the system as well as to identify the parameters that would be used in the next phase of evaluation. In the second phase, we evaluate the approach on real-world data where we implement the middleware and RSUs on Docker containers. The goal of these experiments is to identify the effects of task allocation algorithm on processing delay, accuracy, and generated routes.

5.4.1 Phase 1: Feasibility Test and Parameter Identification

In this section, we evaluate our task allocation algorithm for decentralized architectures in a synthetic environment. We discuss the experimental setup and data used in our simulation. We evaluate different task allocation configurations used in our algorithm and finally we evaluate our approach and compare it to a centralized task allocation solution.

- **Network Graph**: We used HERE API [64] data which includes speed data for each road segment within the Nashville Metropolitan area with a bounding box of $(-87.04999, 35.97, -86.510, 36.42)$. A network graph

is generated from this data which consists of a total of 2623 nodes and 10880 edges. We partition this network graph into equidistant grids using geohashing with a precision of 5 shown in Fig.5.1.

The grid area affects total number of sensors which in turn affects the processing time for each RSU. We may vary the area and number of grids in order to identify the limitations and capabilities of RSUs in maintaining a particular area or number of sensors. For the simulation, we partitioned the Metropolitan area into 613 grids each with an area of $600m^2$.

- **Road Side Units**: RSUs used in this are simulated in a centralized configuration. Each grid above, is assigned to a corresponding RSU. RSUs are assumed to have infinite memory and accept and execute tasks generated by the task allocation algorithm. RSUs are assigned a grid's sub-graph which hold location and speed data for that particular grid.

- **Mobility Data and Accuracy**: Speed data utilizes data provided by the HERE API for the month of March 2018. We assume that as the $MD(g, g')$ increases, the model accuracy decreases by an equivalent factor.

- **Trip Query Data**: To simulate routing queries, we synthetically generate 12,000 source and destination pairs at random from the Nashville Metropolitan Area. A time window was chosen randomly out of 24 hours. An optimal sequence grid is generated from each pair and we perform our task allocation algorithm based on this optimal sequence grid.

Based on the top 10 employers in Nashville which have nearly 140,000 employees [69] and assuming people have variable work shifts with a variance in departure time of one hour, our system needs to be able to process around 12,000 queries every 5 minutes.

We simulate the division of these user queries into tasks and then allocate these tasks to 600 different RSUs. We vary the neighbor levels and identify the effect the distribution of 230,000 tasks (generated from the 12,000 queries) has on overall query response time and model accuracy.

Allocation will depend on the neighbor levels. Trip queries are consistent for all tests, only the task allocation changes. For 0^{th} level task allocation, we force the

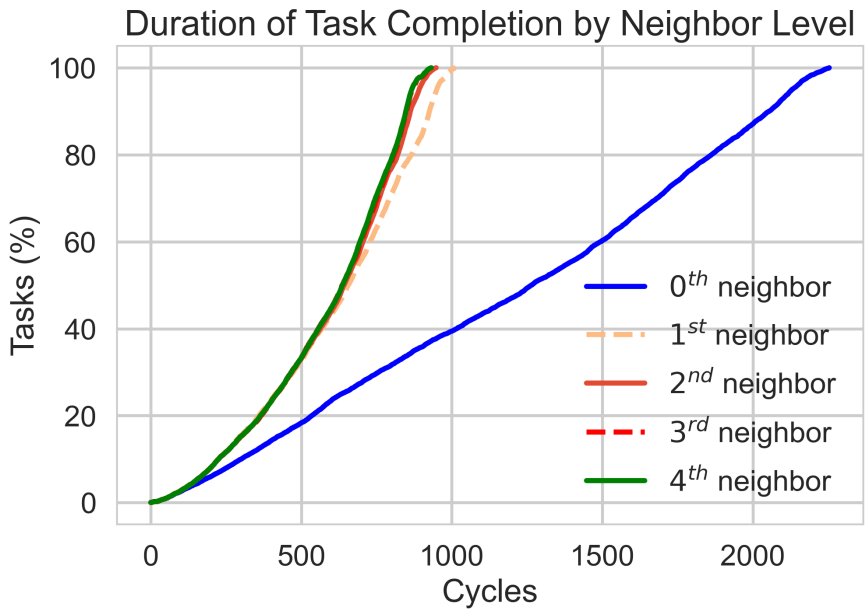

Figure 5.6: Synthetic Processing time for all 12,000 queries. Neighbor Level 0 takes more than 2 times longer than when utilizing neighbor grids

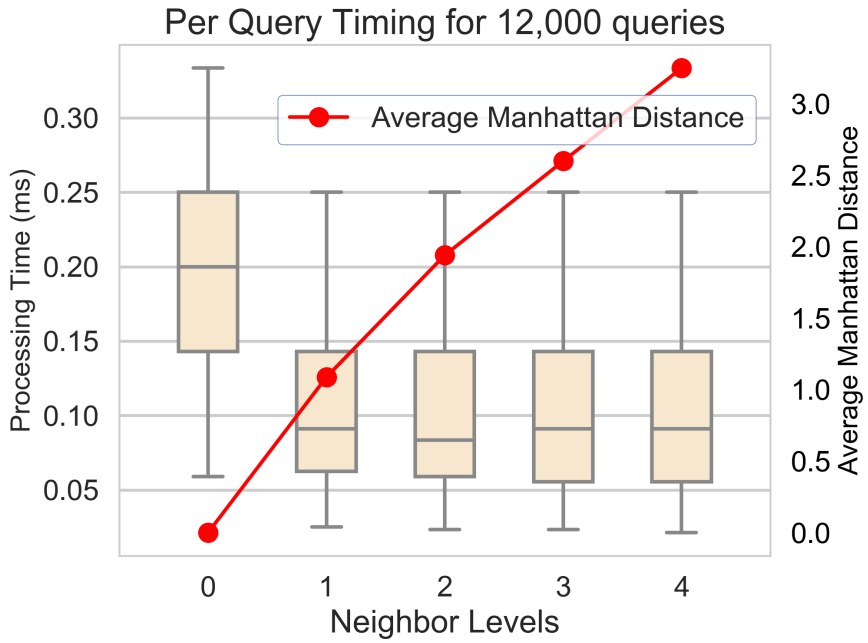

Figure 5.7: The trade-off when utilizing neighboring nodes to decrease query response time is the increase in Manhattan Distance between optimal and allocated RSUs, resulting in a decrease in model accuracy

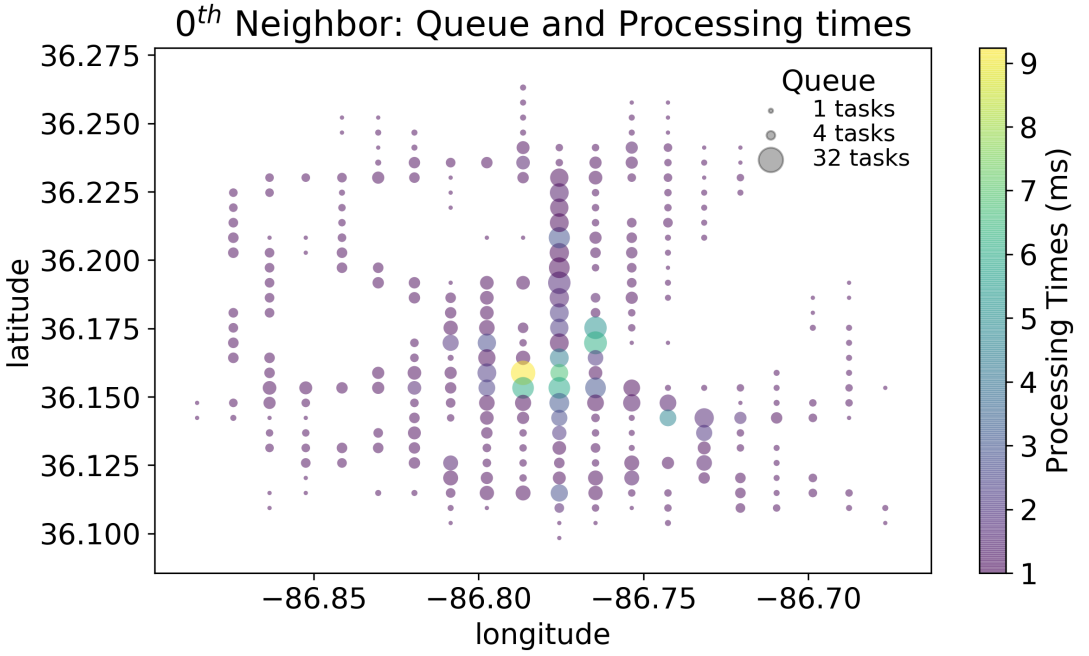

Figure 5.8: 0^{th} Level Neighbors Utilization for the first 100 queries

task to be assigned to the optimal grids SG_q without considering any processing constraints. For the other levels (1^{st} and 2^{nd}) we set constraint and distribute tasks to neighbor nodes if the optimal RSU for use is already over-utilized.

Figure 5.8 shows the limited number of RSUs being used for task allocation. In addition, the RSUs with the heaviest load in the Downtown Nashville areas have much more utilization compared to the surrounding RSUs.

Comparing it to Fig. 5.9, we can see that a larger number of RSUs across the map are being utilized for task allocation. The downtown grids that were over-utilized in Fig. 5.8 still see heavy usage, however the neighboring RSUs allow the system to meet D_{th} by accepting more tasks.

Processing starts once all tasks have been allocated. Processing time is measured in cycles. Total processing time is based on the number of cycles needed before all tasks in all RSU's task queue have been processed. A task is processed by removing it from the RSU's task queue if it is flagged as done, else it is left there to be checked again in the next cycle.

We define each cycle as one loop across all RSUs. Since tasks must be processed

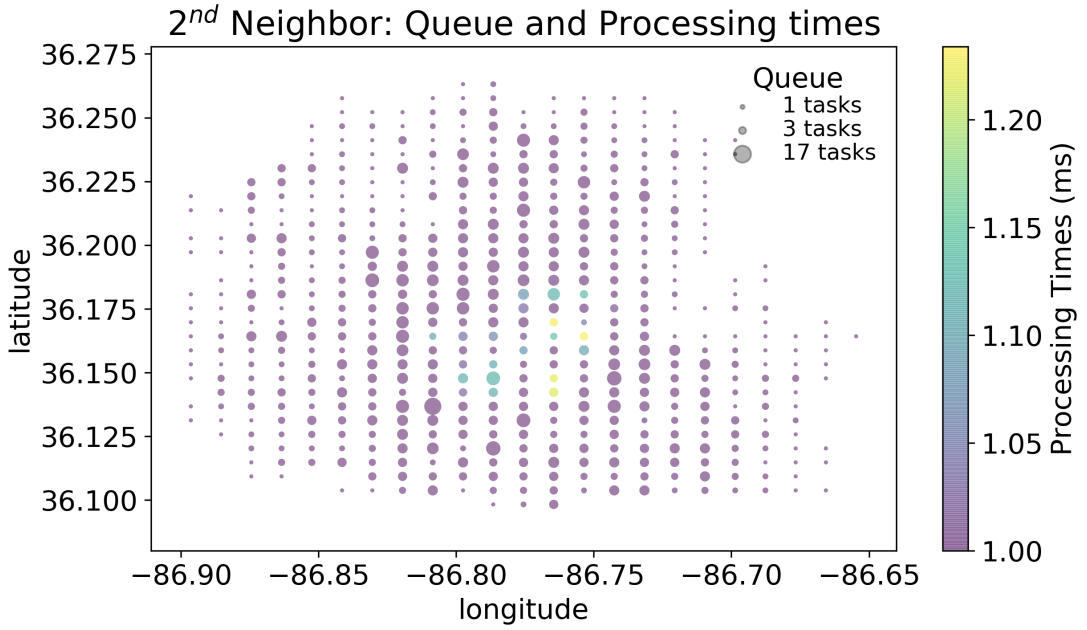

Figure 5.9: 2nd Level Neighbors Utilization for the first 100 queries

in sequence, task $t_{q,i-1}$ must be executed before task $t_{q,i}$. For every cycle, we loop through each RSU with a non-empty task queue and we get its task at the start of the queue and flag it as done if it is the starting task of the query, $t_{q,0}$ or if the previous tasks from the same query $t_{q,i-1}$, has been done. A task that is done is removed from the task queue. A single cycle is done once all RSUs with non-empty task queues have been checked. The cycle repeats as long as there are RSUs with non-empty task queues.

At the end of every cycle, tasks flagged as done are reviewed. A query q is finished when all of its tasks, $T_q = \langle t_{q,1}, t_{q,2}, ..., t_{q,k_q} \rangle$, have been flagged as done.

When forcing tasks to be allocated optimally (level 0), certain RSUs have more than 2000 tasks allocated to it. Increasing the search grid for task allocation allows tasks to be assigned to more RSUs, resulting in a much more balanced distribution of tasks. This is reflected in the total task processing time. Figure 5.6 shows that using at least neighbor level 1, all tasks finish in less than half the cycles of neighbor level 0.

Due to the increase in the number of possible RSUs to assign tasks to, the aver-

age Manhattan distance between the optimal and actual RSUs used increases the higher the neighbor level. Figure 5.7 shows the relationship between Manhattan distance and average cycles per query completion for different neighbor levels.

5.4.2 Phase 2: Real-World Data and Scalability

The goal of this section is to investigate the feasibility of the approach discussed in Phase 1. We compare the performance of our service's task allocation algorithm to the intuitive case of using only the optimal allocations to have the highest model accuracy. We use total query processing time and route travel time accuracy as performance metrics to show the effect of our algorithm on the service.

We perform simulations and experiments on Docker containers to be able to easily test and adjust the parameters of the system and algorithms while also keeping scalability in mind. The number of grids and queries used in this experiment has been reduced since the actual route generating processes will be done on containers on a single physical machine. Table 5.2 shows the devices that were used or tested in this section.

From 600 RSUs, we choose a subset of 49 RSUs. We match this reduction in the number of processing nodes to the number of queries. We pick 1,000 queries from a total of 12,000 queries. This allows us to focus more on understanding how communication between RSUs occurs while maintaining the query to RSU ratio we used in the simulation experiments. For this experiment, we assume that RSUs have the similar computational capacity to Raspberry Pi-like devices.

Container Benchmarking

Since we assume the RSU to be similar to a Raspberry Pi level device, we verify the difference in computational capacities of our Docker containers and a single Raspberry Pi. For the following experiments, we use a single Mac mini hosting all 49 RSUs via Docker containers is used. This was used without any limitations on its computational and memory resources. We run a route planning benchmark on all 49 RSUs simultaneously and then on a single Raspberry Pi. The benchmark runs 1,000 queries on the devices and we get the average time it takes to complete all queries as a measurement of the device's computational capacity.

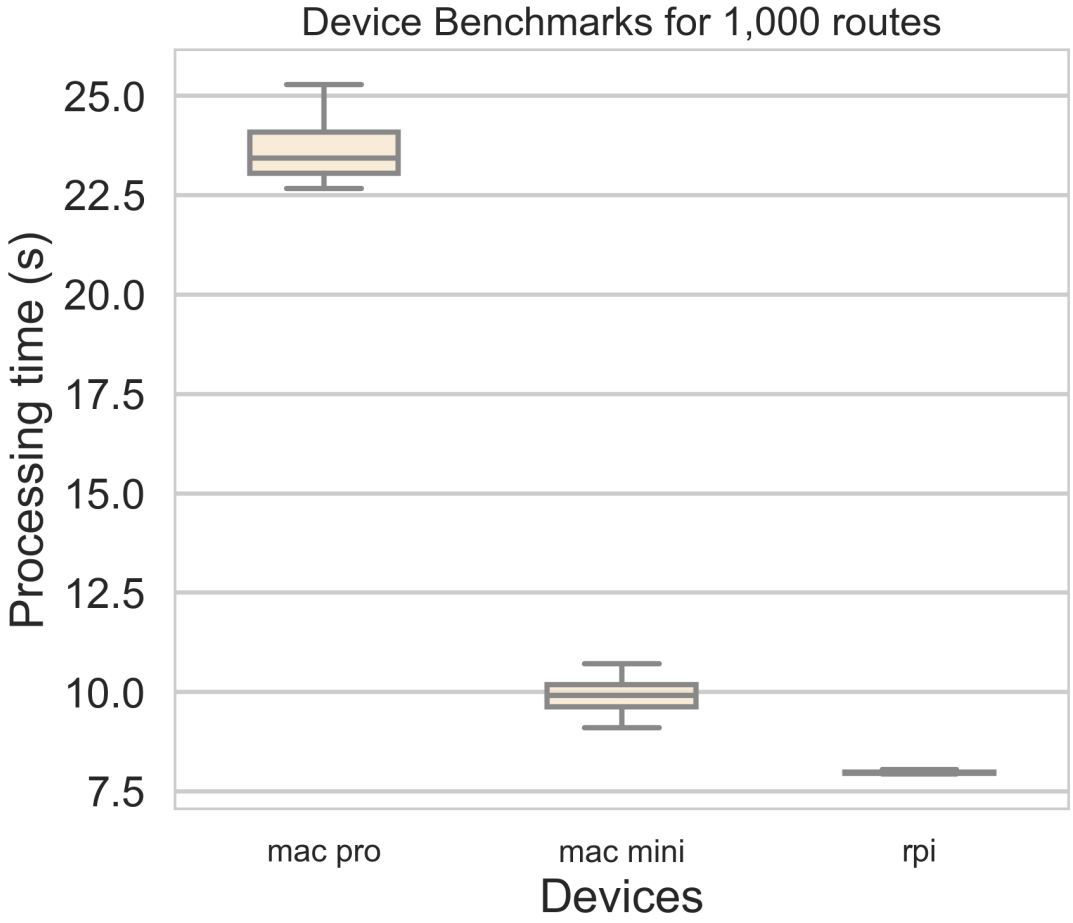

Figure 5.10: Comparing the overall processing time for 1000 trip queries using Docker containers and Raspberry Pi 3B. Mac mini and Macbook Pro devices are running 49 containers simultaneously and benchmarks are run simultaneously on all containers.

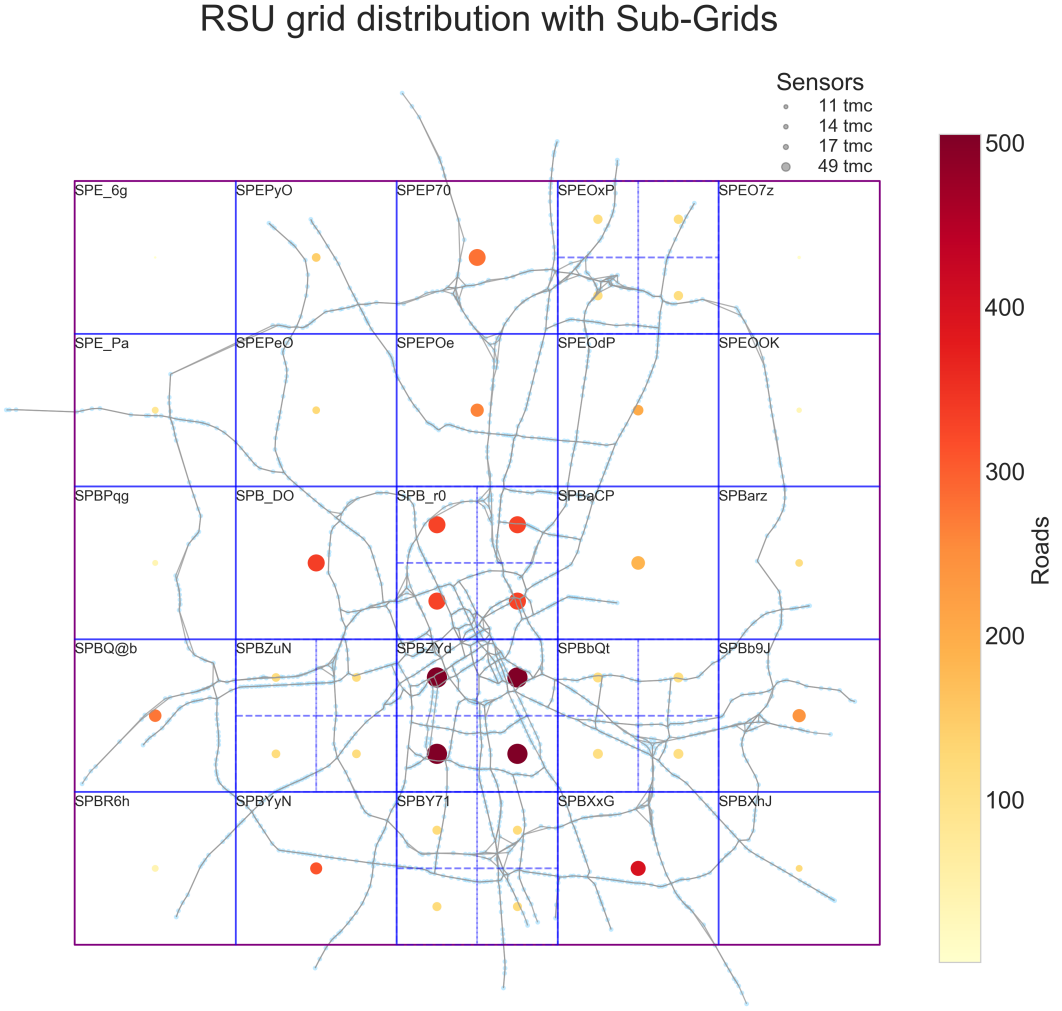

Figure 5.11: The target area is divided into a 5x5 grid layout. Each over-utilized grid is further divided into sub-grids. The bar shows the density of road segments in the grid.

We tested two different devices, a Mac mini, and a Macbook Pro, as the central physical host machines for containers. Figure 5.10 is the result of running route planning benchmarks on the devices.

The results show that the 49 RSUs being emulated on Docker containers on the Mac mini have a similar computational capacity as Raspberry Pi 3B. For all subsequent experiments, a Mac mini hosting all 49 Docker containers emulating RSUs, will be used.

Table 5.2: Comparison of Computational Resources of Devices Used in the Experiment

Device	CPU	RAM
Mac mini	3 GHz Intel Core i5	64 GB DDR4
Mac pro	2.9 GHz Intel Core i7	16 GB DDR3
RPi 3B [70]	1.5GHz Quad core Cortex-A72	1GB LPDDR4-3200
RSU [71]	800 MHz Dual core iMX6	1GB

A sample specification for road side units is included here.

Experiment Parameters

- **Network Graph**: This uses the same network graph as the first part of this experiment. A network graph with 2623 nodes and 10880 edges.
- **Graph Partitions**: The network graph is first divided into a **5 x 5** grid. However as Figure 5.11 shows, some grids still have a higher density of roads and sensors. These grids were then sub-divided into a further **2 x 2** grid. Figure 5.11 shows the division as well as the density of roads and sensors assigned to a particular grid. Each grid has a size of 15 km^2, while each sub-grid is 3.75 km^2.
- **Road Side Units**: 49 RSUs are used and deployed. RSUs are assumed to be static and connected via a wired network.
- **Mobility data**: To simulate traffic across roads, data collected by HERE API for the region on March 2018 is used. These data are collected by sensors placed near road segments. We assume these sensors gather data

and send them to representative RSUs for aggregation and processing. This data is logged in one-minute intervals which are then averaged into one-hour time windows, resulting in 24-speed data entries per RSU per day. These are used as the edge weights for the network graph above.

- **Trip Query data**: From the network graph, we pick 1,000 uniformly distributed source and destination pairs along with randomly selected departure times. We assume these queries are sent within five minutes, aggregated, and then processed.
- **Constraints**: For all the subsequent experiments, TQ_{th} is set at 100.

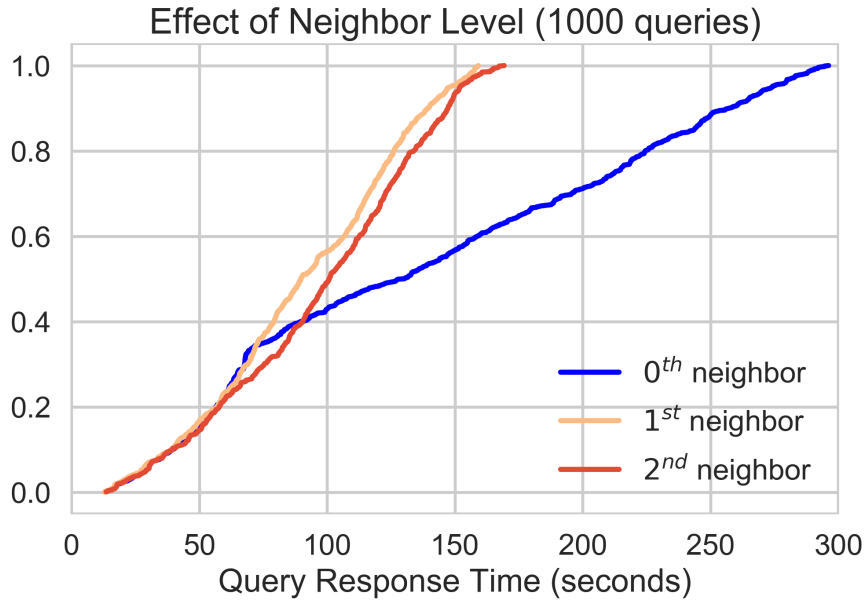

Figure 5.12: CDF curves showing the effect of Neighbor Levels on the query response times

5.4.3 Experiment Evaluation

Processing time, $ct(t', r)$, is measured as the time it takes for a sent query to be responded to with the final route. These include the generation of the optimal sequence grid, task allocation, and task execution. Accuracy is measured in terms of total travel time per route.

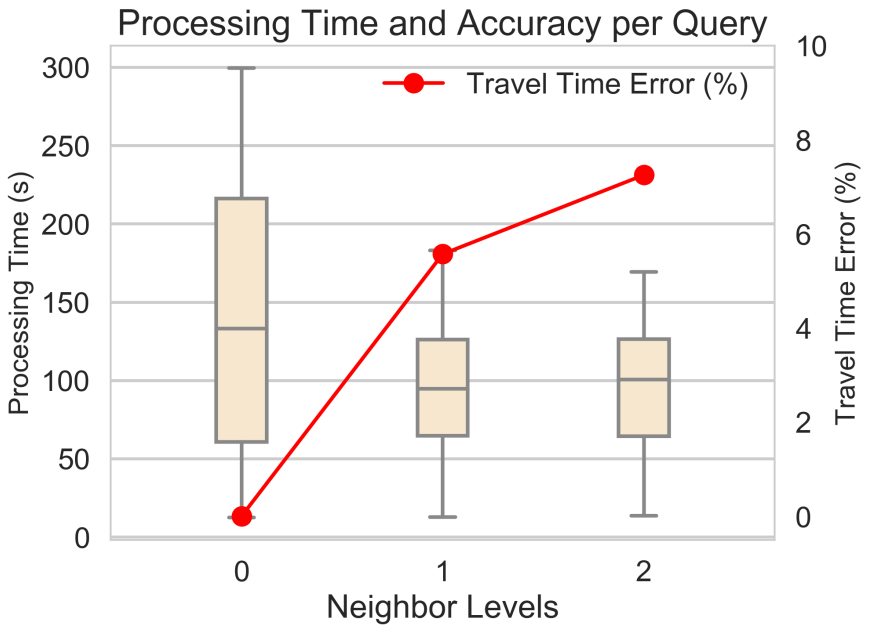

Figure 5.13: Effect of utilizing neighbor nodes during task allocation on total query time

To quantify the effects of varying neighbor levels on task allocation, processing time, and accuracy, we send 1,000 trip queries to the broker. All queries are sent to a single broker and it performs all sequence grid (SG), generation as well as task allocations.

Table 5.3: Effect of Query Count on Allocation and Processing (Neighbor Level 1)

Query Count	Allocation Time	Total Processing Time
100	0.4 s	37 s
200	0.7 s	54 s
500	2 s	111 s
1000	1.7 s	145 s
2000	8.6 s	360 s

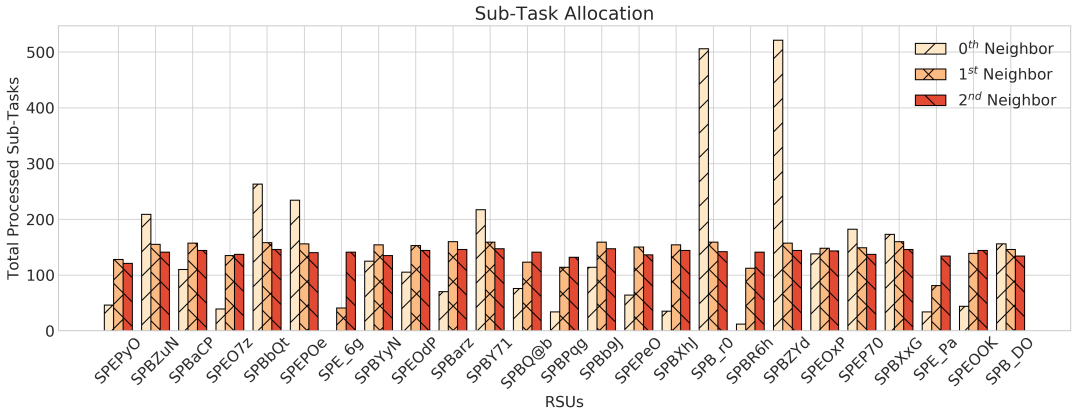

Figure 5.14: Task Allocation for 1,000 queries, each divided into sub-tasks

Task Allocation

Task allocation time is negligible when compared to the total processing time as shown in Table 5.3. Total task allocation for 1000 queries, is 1.7 seconds (the execution time of Algorithms 1 and 3) while total processing time takes 145 seconds.

Figure 5.14 shows the breakdown of how sub-tasks are allocated to the different RSUs. Certain high usage RSUs that have sub-grids have their tasks distributed internally. With neighbor level 0, these high usage RSUs have five times the number of tasks allocated to it while some RSUs such as *SPE_6g* have little to no allocated tasks. In neighbor level 0, there is a direct correlation between the number of tasks and the number of roads in an RSU.

Processing Time

Figure 5.12 shows how much faster processing times are when utilizing neighbor grids. Without using neighbor grids, the service gets overloaded when around 35% of the queries are being processed. On the other hand, using neighbor grids allow the service to comfortably handle subsequent tasks. The similarities between Neighbor levels 1 and 2 are due to the computational limitations of the host device.

The processing time for each query includes the delay it takes for messages to

be sent between a broker and RSUs. An average message between broker and RSU has 0.5KB of information, while a response to user queries has 0.7KB.

Messages sent from broker to RSUs contain the following information:

- Unique Task ID
- RSU ID allocation
- Next RSU in sequence
- Source
- Destination
- Departure time

In their response to Brokers, RSUs add the following to the message:

- Partial Route
- Partial route travel time

Query response messages contain:

- Unique Query id
- Final route
- Final route travel time

Assuming that RSUs and brokers have a wired/wireless connection such as LTE, this message will be sent in milliseconds and thus negligible. Similarly, processed responses will be sent back in milliseconds using Dedicated Short Range Communication (DSRC) or LTE connectivity.

Accuracy

Tasks are processed faster when using neighbor levels 1 and 2. This is because tasks are allocated to less utilized but less optimal RSUs, resulting in a decrease in model accuracy. We measure this accuracy as the overall travel time for a generated route. We assume that routes generated by with neighbor level **0** have 100% accuracy. Each task allocated differently from the most optimal RSU penalizes the data being obtained. This penalty is a function of the Manhattan distance, between optimal and assigned RSU as described in Equation 5.8.

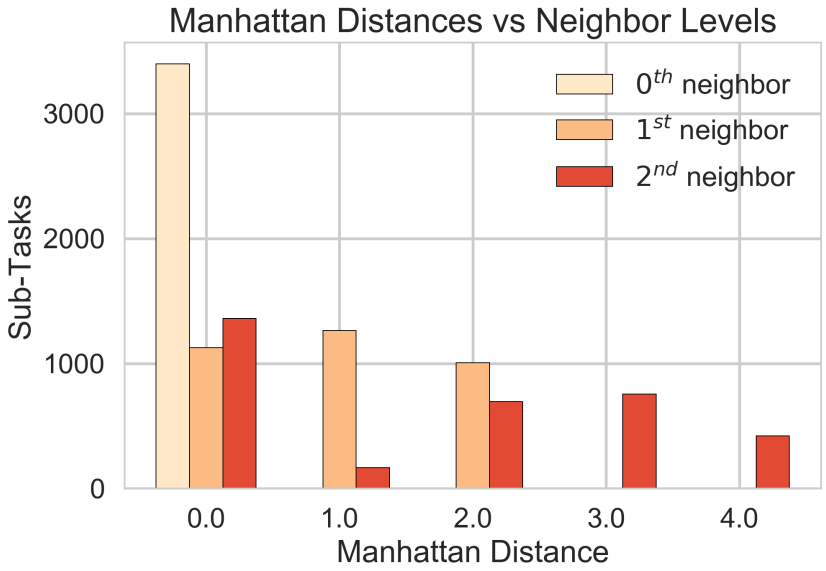

Figure 5.15: Histogram of Manhattan Distances based on Neighbor Levels

To simulate this loss of accuracy, we define that the Manhattan distance between optimal and assigned RSU be equivalent to the staleness of the speed data in minutes. In prior experiments, a Manhattan distance of 1 means that only speed data from 60 minutes ago will be available to that assigned RSU to be used in generating routes.

We assume that this staleness of data is an adequate measure of possible inaccuracies due to sub-optimal task allocations. Figure 5.13 shows that the trade-off for decreasing query response time by around 50% is a 5.5% loss in model accuracy. While Neighbor level 1 is providing an acceptable trade-off, Neighbor level

2 shows no substantial decrease in processing time to justify the additional 2% of loss. This loss in accuracy is primarily due to the Manhattan distance between optimal and assigned RSUs as shown in Figure 5.15. With neighbor level 2, almost 400 tasks that were assigned to an RSU four Manhattan distances away from its optimal allocation.

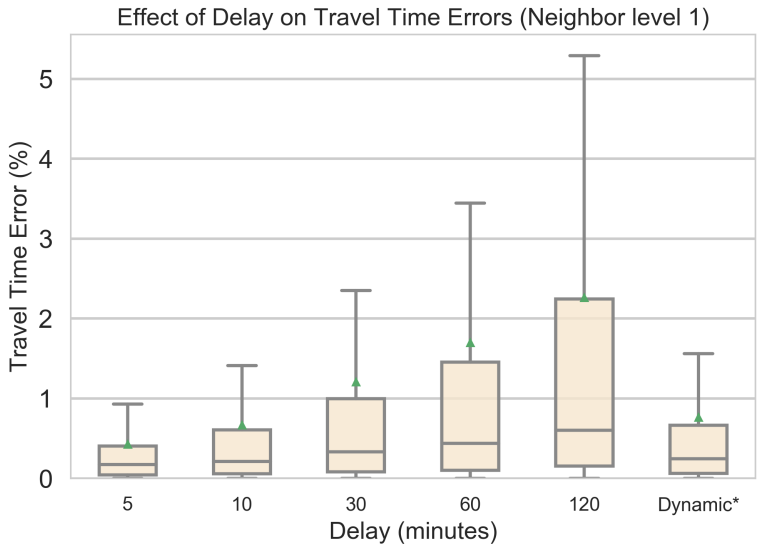

Figure 5.16: Effect of varying delay on average travel time errors. The rightmost point (Dynamic) is based on assigning different delays based on road speed changes. Nodes get speed updates in increments of 5, 10, 30, and 60 minutes based on how often the road speed changes.

The 60-minute delays were based on the assumed speed with which data is propagated to neighbor nodes. We vary the delay of speed updates between neighbor nodes to verify its effect on the overall accuracy of the system. Figure 5.16 shows that as we increase the delay between speed updates, the average travel time error per trip increases. Also, we defined dynamic speed updates based on how often the road speed changes. The goal for this dynamic method of updating speed data is to obtain a balance between the number of messages between RSUs and the overall travel time accuracy. In this method, we divided the roads into four distinct categories, and based on the frequency of speed changes throughout the day, we assign them varying speed updates ranging from 5, 10, 30, and 60 minutes.

This gives a better description of the effect of road speed changes on the travel time. The drawback would be the communication costs between nodes as updates are sent between them.

Concurrent Query Count

We have seen that there is a substantial decrease in processing time when using neighbor levels 1 and 2 over neighbor level 0. However, a median of almost 100 seconds of processing time for neighbor level 1 is still quite large. Here we test varying concurrent queries on the system with a neighbor level 1.

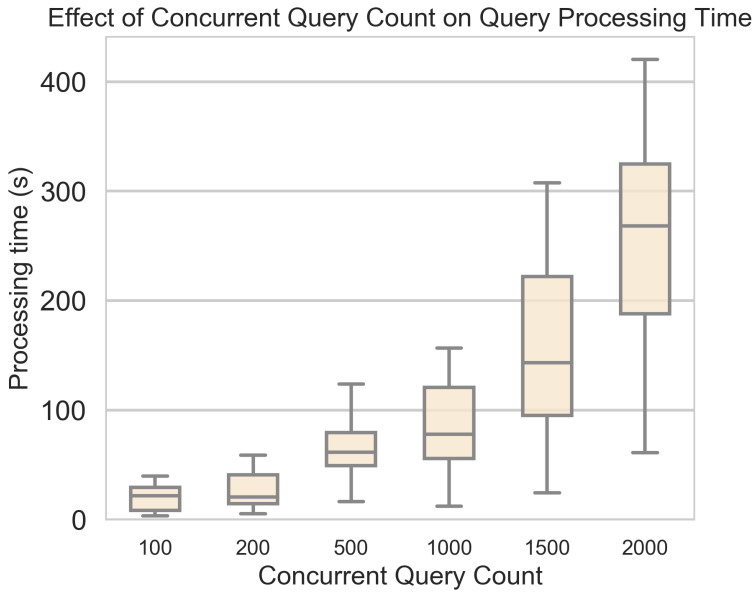

Figure 5.17: Box plots showing the effect of the number of concurrent queries sent on the query response times of the system (Neighbor level 1)

Figure 5.17 shows how long it takes for different concurrent queries to be completely processed. Processing time increases tenfold when increasing the number of concurrent queries from 200 to 2,000. Table 5.3 shows the time it takes different numbers of concurrent queries to be queried, allocated completely, and finally processed.

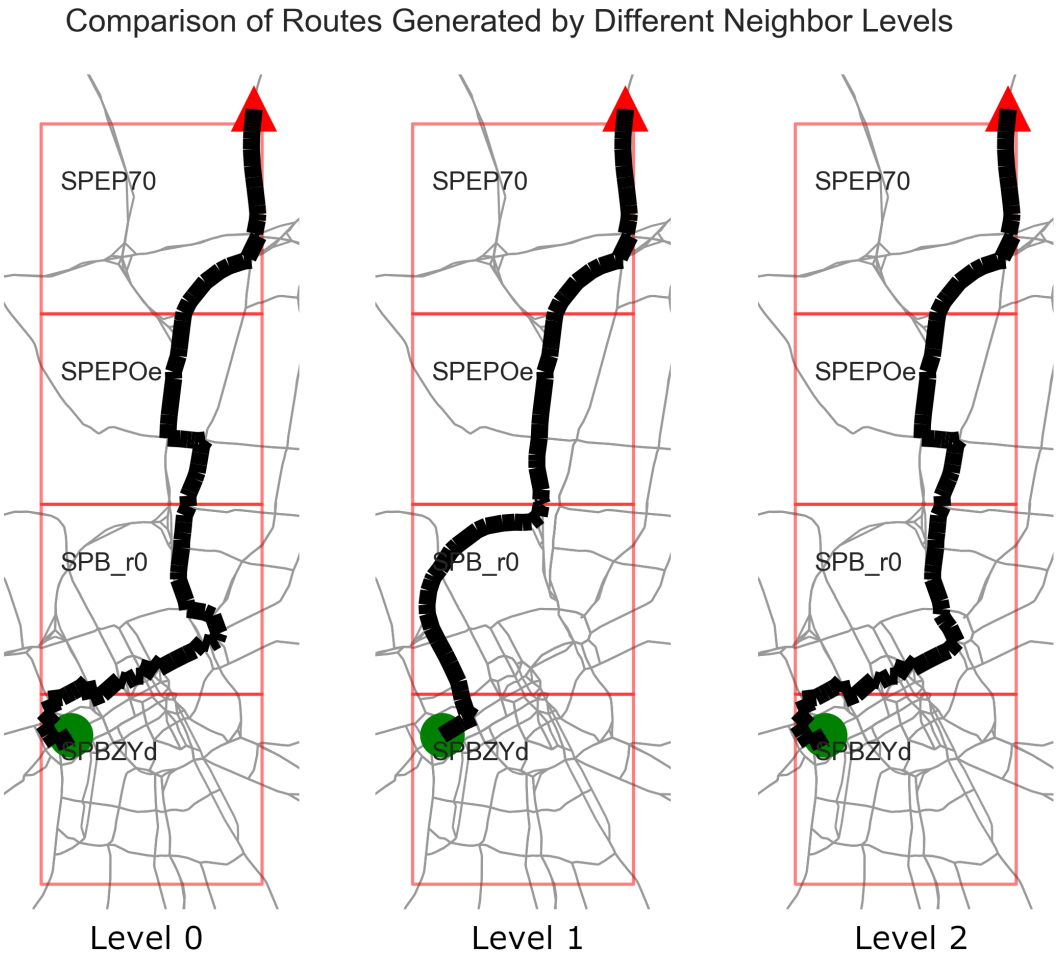

Figure 5.18: Routes generated with varying neighbor levels. Circle and triangle are source and destination respectively. Only the 2nd half of the route is shared by all.

Route Generation

For 1,000 trip queries, the majority of the trips have the same routes regardless of the neighbor level. This is either due to tasks being allocated optimally or speed data is close enough that there is no need for rerouting. These instances occur in more than half of the trips. For these trips the average difference in trip travel times for neighbor levels 1 and 2 are **3.5**% and **4.5**% respectively.

However, for the remaining trips, the routes differ because the staleness of the data was large enough to affect the route planning. For these trips, users had to be routed through different roads. For these trips the average difference in travel times for neighbor levels 1 and 2 are **12.7**% and **18.11**% respectively. Although higher, these differences occurred only in 81 trips out of 1,000.

Figure 5.18 shows a worst-case scenario due to data staleness. Each route passes through the same set of grids; however, the route generated by level 2 diverges the most, which is to be expected, especially on roads where traffic changes more drastically. This particular route passes through downtown, which we assume to have very large speed changes throughout the day. Due to the staleness of available data due to task reallocation, the final route differs between neighbor levels. In this scenario level 1 and 2 had a difference in travel times of **7.37**% and **31.81**% respectively.

5.5 Discussion and Limitations

5.5.1 Discussion

We showed in Figure 5.10 that emulating 49 RSUs on a Mac mini gave similar results to Raspberry Pi-like devices. Running all experiments using this configuration, gives us an idea of how a Raspberry Pi or a similar device would perform as an RSU given the same parameters.

We checked how characteristics of a decentralized route planning service would change depending on varying parameters. In Section 5.4.3, we found that by simply increasing the region of interest for available RSUs, we can decrease the processing time. Figure 5.13 shows that by including neighbor grids as possible reallocation alternatives, user queries are processed **50**% faster compared to not

using neighbor grids.

Section 5.4.3 shows that while this task allocation algorithm offers the benefit of decreased processing time, there is a trade-off with model accuracy. As tasks get allocated farther and farther away from the most optimal RSU, the available speed data becomes staler. However, Figure 5.13 shows that for 1,000 queries the total travel time accuracy is decreased by an average of only **7**%.

Finally, we tested the actual route being generated by the service in Section 5.4.3. We found that utilizing neighbor nodes did not have a large negative effect on the actual route planning algorithm. Out of 1,000 trip queries, more than half was given the same route regardless of the travel time difference which had an average of **4.5**% decrease in model accuracy. Less than 10% of the trips had a route that was affected by the staleness of available data due to the distance between optimal and assigned RSUs. Of these trips, the decrease in accuracy is **12.7**% and **18.11**% for neighbor levels 1 and 2 respectively. The decrease in accuracy does not impact the correctness of the route, only the timeliness of the data being used. Lower quality or less accurate route increases the chances that the route will pass through congested roads.

5.5.2 Limitations

The first limitation of our study was the way we stored local speed data. Depending on the transport network density, speed data could easily reach millions of rows of data. While an updating database of time-series data is preferred, we found that it consumes too much processing time (when accessing stored data files) or waits for too long (database querying) for the response to be sent in close to real-time.

To work around this and still be able to provide speed data for route planning, we averaged data over the month into 1,440 data points (per minute, per day) and stored them as a look-up table. While real-time updating local data provides significant improvements in actual route generation, this is beyond the scope of this study.

The second limitation was the way the optimal sequence grids are generated. We assume that there is only a single optimal route per trip query. For trips with multiple optimal feasible routes, we must update the Equivalent Grid Routing

model ($\hat{E}$). Updating the model to handle such trips and routes is beyond the scope of this dissertation.

5.6 Summary

In this section, we implement a fully realized decentralized route planning service that runs on the middleware described in previous chapters. This service provides the shortest route from a source to a destination. To achieve the highest accuracy, tasks must be allocated to the most optimal RSUs defined by the optimal grid sequence. However, due to resource constraints, allocating all tasks to their optimal RSUs will result in delays to the overall query response.

The problem is how to ensure that all task assignments satisfy query response constraints while maximizing the overall accuracy of the result. To solve this, we proposed a distributed task allocation algorithm that identifies the most optimal assignments to meet accuracy and time requirements.

Evaluating the system both using synthetic simulations and emulated devices with real-world data, we measured the efficiency of the system based on various parameters such as the Neighbor levels, concurrent queries, and computational capacity. We found potential trade-offs between overall query processing time and model accuracy when using different neighbor levels. By having a wider search area for available RSUs, user queries are processed **50**% faster compared to not using neighbor grids with only an average of **7%** decrease in model accuracy.

6 Conclusion

6.1 Summary

In this dissertation, we introduced the Information Flow of Things (IFoT) framework. We presented our implementation of it called IFoT middleware. The motivation for our work was the need for a system that utilizes distributed computing nodes at the edge. The goal was to create a more edge-centric system where smart city services can be deployed to, and function without access to Cloud-based resources and services.

We presented our idea of the IFoT middleware and discussed the architecture needed to realize it. The architecture uses a combination of edge nodes set as brokers and workers. The middleware gathers and processes local data at the nodes maintaining privacy and decreasing latency. Each worker processes tasks that have been preprocessed and distributed by the broker based on various task graphs. Distributed task processing allowed the system to execute queries faster, the greater the number of workers.

To further leverage the capability of the middleware, we deployed two smart city services: Resilient Smart Mobility and Route Planning services. Resilient Smart Mobility aggregates the local data in decentralized nodes and provides traffic information to users such as accidents, hazards and detours. Due to the dangers of data falsification for such a time-sensitive and critical service, we provide an anomaly detection check over the data aggregation.

Route Planning Service delivers a fully implemented service on the middleware. This service needs to provide the shortest route from a source to a destination. Tasks are divided and distributed to various nodes where their local road speed data is used to create a decentralized route which is then aggregated. To achieve the highest accuracy, tasks must be allocated to the most optimal RSUs defined

by the optimal grid sequence. However, due to resource constraints, allocating all tasks to their optimal RSUs will result in delays to the overall query response. To solve this we use a task allocation algorithm that efficiently distributes tasks to the nodes.

We evaluated both systems using a testbed and simulations, we found that using the IFoT middleware allowed services to perform tasks on distributed nodes and their local data. This allowed services to function without the need for Cloud-based resources and services. In addition, this allowed for an increased level of privacy by restricting each node's access to only local data. By using our proposed task allocation algorithm, user queries are processed **50**% faster compared to not using neighbor grids with only an average of **7%** decrease in model accuracy.

In this dissertation, we have presented a working build of the Information Flow of Things middleware. We addressed several requirements of IFoT middleware for smart city services such as architecture, latency, resiliency, scalability and data processing. This are the first steps towards the goal of implementing decentralized smart city services over edge devices. To ensure the viability of our middleware and validity of our idea, we focused first on implementing smart transportation services. While smart transportation service is a single domain within the countless possible smart city services, it serves as an indicator of the middleware's capabilities, running on edge devices that are assumed to be an integral part of smart cities.

6.2 Limitations and Future Work

The goal of the dissertation is to create a middleware that will be deployed over edge devices within a smart city, that would host generic smart city services. We have only discussed and developed the foundations of such a middleware. In order to improve this middleware, dynamic device discovery and more advanced data replication algorithms are needed. While task scheduling and allocation was tackled, it is for a specific smart city use-case, smart transportation. Other services might have different data storage implementations and that devices might be more dynamic. In order to be useful for more generic scenarios, the middleware must also be reliable and resilient. This can be improved by transitioning into a

more broker-less framework. Only then could smart city services for any smart city domain be deployed to the system.

Once these remaining challenges and requirements have been addressed more smart city services can be deployed over it. The current route planning service can be extended for multi-modal transportation systems, which will benefit even more citizens. Smart tourism is another smart city service that can be deployed over the middleware. Smart tourism, relies on aggregating and processing data from multiple sources. Information such as crowdedness, congestion, temperature, social media mentions, can be aggregated, processed and delivered as sightseeing tour recommendations to users. Services such as these make use of the middleware running over distributed edge devices.

There are a few open issues which require additional work. In chapter 3, deployment was only done over RSUs that were simulated using Docker containers. Network connectivity was assumed to be wired and devices were relatively static and the components had a priori knowledge of both the broker and other edge devices. Deployment on real world devices, including mobile edge devices, will be more challenging.

In chapter 4, the goal was to create a tool that allows us to identify which parameters would suit a middleware deployment. However, the graph shown in our results is not a general outcome for all possible middleware deployments. Other middleware deployments might favor certain clustering scenarios over others. The results should not be used as a representation for all frameworks.

With regards to smart transportation, we must verify the performance of the middleware and distributed route planning service. To accomplish that, the system must be deployed and validated with real-world test cases. The actual deployment of this system introduces new challenges that must be solved such as geographical distances and communication delays. Thus, the network configuration and architecture must be carefully planned. In Japan, ITS (Intelligent Transport Systems) are steadily expanding with the popularization of ETC (Electronic Toll Collection) systems. With ETC 2.0 [72], vehicles have the ability for V2V and vehicle to RSU communication. We should consider how our architecture will be deployed in such an environment.

Finally, the decentralized and distributed nature of the edge devices presents an

opportunity to leverage federated learning within its data processing capabilities. The edge devices offer the perfect location for this type of machine learning. In the future, this should also be looked at as an additional method of processing data.

Acknowledgements

This would not work would not have been possible without the support of many people. First, I would like to thank my supervisor, Prof. Keiichi Yasumoto, who provided valuable input and guided me tirelessly, throughout the duration of my stay. I would also like to thank the members in the U.S.-Japan Project specifically Prof. Abhishek Dubey, Prof. Sajal Das, Prof. Hirozumi Yamaguchi and Prof. Shameek Bhattacharjee. They have expanded my knowledge through the constant dialogues and projects we have worked on.

I would also like to thank the present and past professors of the Ubiquitous Computing Laboratory: Associate Prof. Yutaka Arakawa, Assistant Prof. Hirohiko Suwa, Assistant Prof. Manato Fujimoto, Assistant Prof. Yuki Matsuda, Assistant Prof Yugo Nakamura and Assistant Prof. Teruhiro Mizmumoto, all of whom gave valuable comments and contributed ideas not only for my research but for my life here in Japan.

I also thank Prof. Kazutoshi Fujikawa for his valuable input and comments that helped clarify and improve the study.

I would also like to thank the laboratory's secretaries, Ms. Nao Yamauchi, Ms. Megumi Kanaoka, and Ms. Eri Ogawa, who handled all the administrative matters during my stay here in Japan, and who were always very gracious despite the language barrier.

I would also like to thank all the members of the Ubiquitous Computing Systems Laboratory for all their friendship, support and for all the fun social gatherings.

*
* *

I would like to thank my own family for their love and support.

References

[1] Cisco delivers vision of fog computing to accelerate value from billions of connected devices | the network.

[2] GSMA. The mobile economy 2020.

[3] Mark Weiser. The computer for the 21 st century. *Scientific american*, 265(3):94–105, 1991.

[4] France Bélanger and Robert E Crossler. Privacy in the digital age: a review of information privacy research in information systems. *MIS quarterly*, pages 1017–1041, 2011.

[5] Kevin Granville. Facebook and cambridge analytica: What you need to know as fallout widens. `https://www.nytimes.com./2018/03/19/technology/facebook-cambridge-analytica-explained.html`.

[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[7] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, 2015.

[8] Sam Lucero et al. Iot platforms: enabling the internet of things. *Whitepaper*, 2016.

[9] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and

Etienne Riviere. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45(5):37–42, September 2015.

[10] Keiichi Yasumoto, Hirozumi Yamaguchi, and Hiroshi Shigeno. Survey of Real-time Processing Technologies of IoT Data Streams. *Journal of Information Processing*, 24(2):195–202, 2016.

[11] Y. Nakamura, T. Mizumoto, H. Suwa, Y. Arakawa, H. Yamaguchi, and K. Yasumoto. In-situ resource provisioning with adaptive scale-out for regional iot services. In *Proceedings of the Third ACM/IEEE Symposium on Edge Computing (SEC 2018)*, pages 203–213, 2018.

[12] H. El-Sayed, S. Sankar, M. Prasad, D. Puthal, A. Gupta, M. Mohanty, and C. Lin. Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment. *IEEE Access*, 6:1706–1717, 2018.

[13] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019.

[14] A. Davis, J. Parikh, and W. E. Weihl. Edgecomputing: Extending enterprise applications to the edge of the internet. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, pages 180–187, New York, NY, USA, 2004. ACM.

[15] Deze Zeng, Lin Gu, Song Guo, Zixue Cheng, and Shui Yu. Joint Optimization of Task Scheduling and Image Placement in Fog Computing Supported Software-Defined Embedded System. *IEEE Transactions on Computers*, 65(12):3702–3712, 2016.

[16] Minh Quang Tran, Duy Tai Nguyen, Van An Le, Duc Hai Nguyen, and Tran Vu Pham. Task Placement on Fog Computing Made Efficient for IoT Application Provision. *Wireless Communications and Mobile Computing*, 2019, 2019.

[17] Abhishek Dubey, Subhav Pradhan, Douglas C Schmidt, Sebnem Rusitschka, and Monika Sturm. The role of context and resilient middleware in next generation smart grids. In *M4IoT@ Middleware*, pages 1–6, 2016.

[18] Subhav Pradhan, Abhishek Dubey, Shweta Khare, Saideep Nannapaneni, Aniruddha Gokhale, Sankaran Mahadevan, Douglas C Schmidt, and Martin Lehofer. Chariot: Goal-driven orchestration middleware for resilient iot systems. *ACM Transactions on Cyber-Physical Systems*, 2(3):16, 2018.

[19] Subhav M Pradhan, Abhishek Dubey, Aniruddha Gokhale, and Martin Lehofer. Chariot: A domain specific language for extensible cyber-physical systems. In *Proceedings of the workshop on domain-specific modeling*, pages 9–16. ACM, 2015.

[20] Ayed Salman, Imtiaz Ahmad, and Sabah Al-Madani. Particle swarm optimization for task assignment problem. *Microprocessors and Microsystems*, 26(8):363–371, 2002.

[21] Qian Zhu and Gagan Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. *IEEE Transactions on Services Computing*, 5(4):497–511, 2012.

[22] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[23] Danilo Ardagna, Michele Ciavotta, and Mauro Passacantando. Generalized nash equilibria for the service provisioning problem in multi-cloud systems. *IEEE Transactions on Services Computing*, 10(3):381–395, 2015.

[24] Charlie Catlett, William E Allcock, Phil Andrews, Ruth Aydt, Ray Bair, Natasha Balac, Bryan Banister, Trish Barker, Mark Bartelt, Pete Beckman, et al. Teragrid: Analysis of organization, system architecture, and middleware enabling new types of applications. Technical report, IOS press, 2008.

[25] Dominik Schafer, Janick Edinger, Justin Mazzola Paluska, Sebastian VanSyckel, and Christian Becker. Tasklets:" better than best-effort" computing. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–11. IEEE, 2016.

[26] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Resource provisioning for iot services in the fog. In *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*, pages 32–39. IEEE, 2016.

[27] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards qos-aware fog service placement. In *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*, pages 89–96. IEEE, 2017.

[28] Jinlai Xu, Balaji Palanisamy, Heiko Ludwig, and Qingyang Wang. Zenith: Utility-aware resource allocation for edge computing. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 47–54. IEEE, 2017.

[29] Jose Paolo Talusan, Francis Tiausas, Keiichi Yasumoto, Michael Wilbur, Geoffrey Pettet, Abhishek Dubey, and Shameek Bhattacharjee. Smart transportation delay and resiliency testbed based on information flow of things middleware. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 13–18. IEEE, 2019.

[30] Z. Ning, F. Xia, N. Ullah, X. Kong, and X. Hu. Vehicular social networks: Enabling smart mobility. *IEEE Communications Magazine*, 55(5):16–55, May 2017.

[31] Road bureau - mlit ministry of land, infrastructure, transport and tourism. `http://www.mlit.go.jp/road/road_e/p1_its.html`. (accessed 03.08.19).

[32] C. Samal, F. Sun, and A. Dubey. Speedpro: A predictive multi-model approach for urban traffic speed estimation. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, May 2017.

[33] S. Eisele, I. Mardari, A. Dubey, and G. Karsai. Riaps: Resilient information architecture platform for decentralized smart systems. In *2017 IEEE 20th*

International Symposium on Real-Time Distributed Computing (ISORC), pages 125–132, May 2017.

[34] M. Mukherjee, R. Matam, L. Shu, L. Maglaras, M. A. Ferrag, N. Choudhury, and V. Kumar. Security and privacy in fog computing: Challenges. *IEEE Access*, 5:19293–19304, 2017.

[35] S. Bhattacharjee, M. Salimitari, M. Chatterjee, K. Kwiat, and C. Kamhoua. Preserving data integrity in iot networks under opportunistic data manipulation. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 446–453, Nov 2017.

[36] Gerhard P Hancke, Gerhard P Hancke Jr, et al. The role of advanced sensing in smart cities. *Sensors*, 13(1):393–425, 2012.

[37] Michael Batty, Kay W Axhausen, Fosca Giannotti, Alexei Pozdnoukhov, Armando Bazzani, Monica Wachowicz, Georgios Ouzounis, and Yuval Portugali. Smart cities of the future. *The European Physical Journal Special Topics*, 214(1):481–518, 2012.

[38] Hafedh Chourabi, Taewoo Nam, Shawn Walker, J Ramon Gil-Garcia, Sehl Mellouli, Karine Nahon, Theresa A Pardo, and Hans Jochen Scholl. Understanding smart cities: An integrative framework. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 2289–2297. IEEE, 2012.

[39] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[40] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[41] Fangzhou Sun, Abhishek Dubey, and Jules White. Dxnat—deep neural networks for explaining non-recurring traffic congestion. In *Big Data (Big Data), 2017 IEEE International Conference on*, pages 2141–2150. IEEE, 2017.

[42] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[43] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[44] Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

[45] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[46] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. In *European Symposium on Algorithms*, pages 804–816. Springer, 2006.

[47] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[48] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[49] Gabriele Di Stefano, Alberto Petricola, and Christos Zaroliagis. On the implementation of parallel shortest path algorithms on a supercomputer. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 406–417. Springer, 2006.

[50] Yuxin Tang, Yunquan Zhang, and Hu Chen. A parallel shortest path algorithm based on graph-partitioning and iterative correcting. In *2008 10th*

IEEE International Conference on High Performance Computing and Communications, pages 155–161. IEEE, 2008.

[51] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.

[52] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[53] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[54] Lin Wang, Lei Jiao, Jun Li, Julien Gedeon, and Max Mühlhäuser. Moera: Mobility-agnostic online resource allocation for edge computing. *IEEE Transactions on Mobile Computing*, 2018.

[55] M. Breitbach, D. Schäfer, J. Edinger, and C. Becker. Context-aware data and task placement in edge computing environments. In *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*, pages 1–10, 2019.

[56] S. Bhave, M. Tolentino, H. Zhu, and J. Sheng. Embedded middleware for distributed raspberry pi device to enable big data applications. In *2017 IEEE International Conference on Computational Science and Engineering (CSE)*, volume 2, pages 103–108, July 2017.

[57] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivière. On using micro-clouds to deliver the fog. *IEEE Internet Computing*, 21(2):8–15, Mar 2017.

[58] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee. A container-based edge cloud paas architecture based on raspberry pi clusters. In *2016 IEEE 4th*

International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), pages 117–124, Aug 2016.

[59] X. Wang, S. Jiang, X. Xu, Z. Wu, and Y. Tao. A raspberry pi and lxc based distributed computing testbed. In *2016 6th International Conference on Digital Home (ICDH)*, pages 170–174, Dec 2016.

[60] S. Egger, T. Hossfeld, R. Schatz, and M. Fiedler. Waiting times in quality of experience for web based services. In *2012 Fourth International Workshop on Quality of Multimedia Experience*, pages 86–96, July 2012.

[61] Rudolf Giffinger, Christian Fertner, Hans Kramar, Evert Meijers, et al. City-ranking of european medium-sized cities. *Cent. Reg. Sci. Vienna UT*, pages 1–12, 2007.

[62] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70–95, 2016.

[63] S. Bhattacharjee, A. Thakur, and S. K. Das. Towards fast and semi-supervised identification of smart meters launching data falsification attacks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 173–185, June 2018.

[64] Here api. `https://developer.here.com/`, 2018.

[65] Huibo Bi and Erol Gelenbe. A survey of algorithms and systems for evacuating people in confined spaces. *Electronics*, 8(6):711, 2019.

[66] Erol Gelenbe, Esin Seref, and Zhiguang Xu. Simulation with learning agents. *Proceedings of the IEEE*, 89(2):148–157, 2001.

[67] A. . Kermarrec, L. Massoulie, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, 2003.

[68] M. Wilbur, C. Samal, J. P. Talusan, K. Yasumoto, and A. Dubey. Time-dependent decentralized routing using federated learning. In *2020 IEEE 23rd*

International Symposium on Real-Time Distributed Computing (ISORC), pages 56–64, 2020.

[69] Department of Finance. Comprehensive Annual Financial Report For the Year Ended. `https://www.nashville.gov/Portals/0/SiteContent/Finance/docs/CAFR/CAFR%202019.pdf`, 2019. (accessed 05.21.20).

[70] Raspberry Pi Foundation. Raspberry Pi 3B Tech Specs. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`, 2019. (accessed 05.22.20).

[71] Siemens. Sitraffic Vehicle2X. `https://www.mobility.siemens.com/global/en/portfolio/road/connected-mobility-solutions/sitraffic-vehicle2x.html`, 2020. (accessed 05.22.20).

[72] Transport Ministry of Land, Infrastructure and Tourism. ITS. `https://www.mlit.go.jp/road/road_e/p1_its.html`. (accessed 08.21.20).

Publication List

Peer Review Journal Paper

1. Jose Paolo V. Talusan, Michael Wilbur, Abhishek Dubey, and Keiichi Yasumoto: "Route Planning through Distributed Computing by Road Side Units," in IEEE Access, vol. 8, pp. 176134-176148, 2020, doi: 10.1109/ACCESS.2020.3026677.

International Conference

1. Jose Paolo Talusan, Yugo. Nakamura, Teruhiro Mizumoto and Keiichi Yasumoto, "Near Cloud: Low-cost Low-Power Cloud Implementation for Rural Area Connectivity and Data Processing." Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), pp. 622-627, 2018.

2. Jose Paolo Talusan, Francis Tiausas, Sopicha Stirapongsasuti, Yugo Nakamura, Teruhiro Mizumoto and Keiichi Yasumoto, "Evaluating Performance of In-Situ Distributed Processing on IoT Devices by Developing a Workspace Context Recognition Service." In Proceedings of the 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 633-638, 2019.

3. Jose Paolo Talusan, Francis Tiausas, Keiichi Yasumoto, Michael Wilbur, Geoffrey Pettet, Abhishek Dubey, Shameek Bhattacharjee: "Smart Transportation Delay and Resiliency Testbed Based on Information Flow of Things Middleware." In Proceedings of the 2019 IEEE International Conference on Smart Computing (SMARTCOMP 2019), pp. 13-18, 2019.

4. Jose Paolo Talusan, Michael Wilbur, Abhishek Dubey, and Keiichi Yasumoto: "On Decentralized Route Planning Using the Road Side Units as Computing Resources," In Proceedings of the 2020 IEEE International Conference on Fog Computing (ICFC 2020), pp. 1-8, 2020.

Other Publications

1. Yugo Nakamura, Yoshinori Umetsu, Jose Paolo Talusan, Keiichi Yasumoto, Wataru Sasaki, Masashi Takata, and Yutaka Arakawa: "Multi-Stage Activity Inference for Locomotion and Transportation Analytics of Mobile Users." In Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers (UbiComp '18). pp. 1579–1588, 2018.

2. Jose Paolo Talusan: "Distributed Processing Middleware on Mesh Network for Connectivity Challenged Environments." 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 457-458, 2019.

3. Michael Wilbur, Chinmaya Samal, Jose Paolo Talusan, Keiichi Yasumoto and Abhishek Dubey: "Time-dependent Decentralized Routing using Federated Learning." Proceedings of the 2020 IEEE 23nd International Symposium on Real-Time Distributed Computing (ISORC), pp. 56-64, 2020.

4. (accepted) Yugo Nakamura, Jose Paolo Talusan, Teruhiro Mizumoto, Hirohiko Suwa, Yutaka Arakawa, Hirozumi Yamaguchi and Keiichi Yasumoto: "ProceThings: Data Processing Platform with In-situ IoT Devices for Smart Community Services." International Workshop on Mobile Ubiquitous Systems, Infrastructures, Communications and AppLications (MUSICAL 2021)