Spring 2018

Leveraging Resources on Anonymous Mobile Edge Nodes

Ahmed Salem
Old Dominion University

LEVERAGING RESOURCES ON ANONYMOUS MOBILE EDGE NODES

by

Ahmed Salem
B.S. May 2006, Arab Academy for Science and Technology, Egypt
M.S. May 2012,Arab Academy for Science and Technology, Egypt

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2018

Approved by:

Tamer Nadeem (Director)

Ravi Mukkamala (Co-Director)

Stephan Olariu (Member)

Dimitrie Popescu (Member)

ABSTRACT

LEVERAGING RESOURCES ON ANONYMOUS MOBILE EDGE NODES

Ahmed Salem
Old Dominion University, 2018
Director: Dr. Tamer Nadeem
Co-Director: Dr. Ravi Mukkamala

Smart devices have become an essential component in our lives. The rapid rise of smartphones, IoTs, and wearable devices has enabled applications that were not possible few years ago, e.g., health monitoring and online banking. Meanwhile, smart sensing has laid the infrastructure for smart homes and smart cities. The intrusive nature of smart devices has granted access to huge amounts of raw data. Researchers have seized the moment with complex algorithms and data models used to process the data over the cloud in order to extract as much information as possible. However, with the pace and amount of data generation, in addition to the networking protocols transmitting data to cloud servers only 20% of what has been generated on the edge of the network was processed. On the other hand, smart devices carry a large set of resources, e.g., CPU, memory, and camera, and these resources sit idle most of the time. Studies have shown that, for much of the time, resources are either idle, e.g., sleeping and eating, or underutilized, e.g. inertial sensors during phone calls. These findings articulate a problem in processing large data sets, while having idle resources in close proximity. In this dissertation, we propose harvesting underutilized edge resources then using them in processing the huge amount of data generated, and currently wasted, through applications running at the edge of the network.

We propose to flip the concept of cloud computing; instead of sending massive amounts of data for processing over the cloud, we distribute lightweight applications to process data on users' smart devices. We envision that this approach will enhance the network's bandwidth, grant access to larger datasets, provide low latency responses, and most importantly include user's contextual information in processing. However, such benefits come with a set of challenges: How to locate suitable resources? How to match resources with data providers? How to inform resources about what to do and when to do it? How to orchestrate applications' execution on multiple devices? and How to communicate between devices at the edge?

Communication between devices at the edge uses different parameters in terms of device mobility, topology, and data rate. Standard protocols, e.g., Wi-Fi or Bluetooth, were not designed for edge computing, hence, they do not offer a perfect match. Edge computing requires a lightweight protocol that provides quick device discovery, a decent data rate, and multicasting to devices in the proximity. Bluetooth features wide acceptance within the IoT community; however, its low data rate and unicast communication limits its use on the edge. Despite its being the most suitable communication protocol for edge computing and despite the fact that it is unlike other protocols, Bluetooth uses a closed source code that blocks its lower layer from all forms of research study, enhancement, and customization. Hence, we offer an open source version of Bluetooth and then we customize it for edge computing applications.

In this dissertation, we propose Leveraging Resources on Anonymous Mobile Edge Nodes (*LAMEN*), a three-tier framework in which edge devices are clustered by proximity. On finding an application to execute, *LAMEN* clusters discover and allocate resources, share the application's executable with resources, and estimate incentives for each participating resource. In a cluster, a single head node, i.e. the mediator, is responsible for resource discovery and allocation. Mediators orchestrate cluster resources and present them as a virtually large homogeneous resource. For example, two devices, each offering either a camera or a speaker are presented outside the cluster as a single device with both camera and speaker; this can be extended to any combination of resources. Then, the mediator handles applications' distribution within a cluster, as needed. Also, we provide a communication protocol that is customizable to the edge environment and the application's need. Pushing lightweight applications that end devices can execute over their locally generated data has the following benefits: First, it avoids sharing user data with a cloud server, which is a privacy concern for many users; Second, it introduces mediators as local cloud controllers, closer to the edge; Third, it hides the user's identity behind the mediators; and finally, it enhances bandwidth utilization by keeping the raw data at the edge and by transmitting processed information. Our evaluation shows both an optimized resource lookup and application assignment schemes, in addition to scalability in handling networks with a large number of devices. In order to overcome the communication challenges, we provide an open source communication protocol that we customize for edge computing applications; however, it can be used beyond the scope

of *LAMEN*. Finally, we present three applications to show how *LAMEN* enables various application domains on the edge of the network.

In summary, we propose a framework to orchestrate underutilized resources at the edge of the network and use them in processing data generated in their proximity. Using the approaches explained later in the dissertation, we show how *LAMEN* enhances the performance of applications and enables a new set of applications that were not previously feasible.

ACKNOWLEDGEMENTS

All Praise is Due to Allah (God)

To my father Prof. Hesham Salem; my mother Nagwa; my siblings H. Elbarawy, Nourane, Omar, & Youssef; and my kids Hesham & Yaseen. This would have not been possible without your unlimited support and believe in me.

To my wife, Sarra, for enduring my miserable work-family balance and long nights with love.

To my advisors, especially Dr. Tamer Nadeem. We had plenty of ups and downs throughout the journey, I certainly learned a lot and will keep learning.

To my mentors Prof. Hussein Abdel-Wahab (RIP) and Prof. Stephan Olariu. I cannot be more grateful for the support I received from both of you. All our one-on-one talks kept me on track or I would have been another dropout.

To the Muslim Student Association (MSA) and the Egyptian Student Association (ESA), especially Dr. Samy El-Tawab and Dr. Ahmed AlSum. Your help during my early days and on made my boarding process so smooth.

To all I have mentioned, as well as those that I have not:
Thank You !

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In 2016, seven billion new smart devices were sold, e.g., smartphones and IoTs, an increase of 30% from 2015 [2, 3], each holding large number of resources, e.g., sensors or CPUs. IoTs alone are estimated to capture more than 200EB (200 million terabytes) of data annually, since inertial sensors are always on [4]. Using cloud computing, it was estimated that less than 20% of this raw data was touched, i.e., stored or processed [5, 6]. Moreover, analysts envision that by 2020 the increase in data generation will double the increase in bandwidth, further amplifying the communication overhead [7, 8]. Hence, cloud computing falls short in processing this large amount of data on backend servers. For example, smart vehicles are expected to generate tons of traffic pattern data, which has to be uploaded to a cloud server for traffic optimization analysis. Not only does this overwhelms the network bandwidth and backend resources, but it overlooks a large amount of resources, including the computational resources onboard of vehicles that can be orchestrated for traffic optimization and for many other applications as we will explain later.

Cloud computing solutions like Amazon EC2 [9], the classical host for these applications, comes with a set of drawbacks: the need for continuous reliable Internet connection, which is not guaranteed and can be expensive if using cellular service; long setup time for cloud server instances; the high payment granularity (for example, a short time job, e.g., single voice recognition, will hold cloud resources for the whole hour and will require full payment); and misusing the network's bandwidth with massive raw data upload. Hence, cloud computing might not be the perfect solution. We target an orchestration of resources at the edge of the network towards executing applications beyond the capabilities of a single device, instead of uploading data and computation to a cloud server. However, not only are end/smart devices are far from hosting big tasks, but they were designed to work individually serving their owners.

Smart devices are continuously receiving software and hardware upgrades in terms of network connectivity, computational, and sensing capabilities. Despite their massive availability and their tie to their end users, they are still limited to simple tasks

serving a single owner including social media and personal sensing, leaving bigger tasks for cloud computing. For example, iPhone, a top notch smartphone, carrying state-of-the-art computational, connectivity, and sensing resources fails to execute voice recognition, i.e., Siri, locally, and offloads it to the cloud for lack of sufficient computation (100x regular queries) [10, 11]. Moreover, smart devices are proven to have idle resources for a significant amount of time, i.e., inertial sensors during a phone call [12].

Smart devices' market penetration, along with their excessive data generation, are accompanied by a shift in the way people think. In the last decade, it was not quite natural for people to share their properties with others. Nowadays, we witness people accepting to share even their personal stuff for a reward. *Uber* is an application that allows car owners to offer paid rides in their personal cars [13]. *AirBnB*, enables offering a room in a house for rent [14]. Moreover, *AirPnP*, enables people to offer their bathrooms for short term rent in return of a small fee, e.g., $1-5 [15]. Further, in the IT domain it is now likely that users can expose their Internet through tethering it to unconnected ones. Many solutions have been developed to organize this process in terms of reward calculation and bandwidth distribution, (ex. Karma Wi-Fi [16]). In short, members of the public nowadays are open to sharing their unused personal things if they are guaranteed a reasonable reward and a reliable security metric.

Edge computing, then, emerges as the natural evolution of such behavioral change to explore the large amount of untouched data at the edge, using many of the idle resources on smart devices. Edge computing is a new paradigm that addresses the explosive growth in mobile data growth by pushing computation at the network edge close to the data sources. Compared to cloud computing which transports all mobile data to cloud servers for processing, edge computing reduces delay, network bandwidth, and mobile device energy consumption. Most existing edge computing approaches use computing resources on the static edge network infrastructure instead of mobile devices. On the edge, however, the network infrastructure will not keep up with the explosive edge data growth and real time application requirements [4, 5].

The collective computing, storage, connectivity, and sensing resources of network edge mobile devices such as smartphones, wearables, and vehicles, offer a new opportunity to extend the boundaries of the network edge closer to the data sources, not only for computation but also for other capabilities such as sensing, storage, and connectivity. The IoT phenomenon amplifies the untapped potential that the fuels

rapid proliferation of edge devices in multiple forms.

The research conducted on the edge so far has been limited to using a standard communication protocols, e.g., Wi-Fi or Bluetooth, none of which is designed to support the quick mobility and low latency required on the edge. Bluetooth, with the advantage of being accepted by the IoT community, has an advantage; however, it is limited with slow device discovery, a low data rate, and unicast communication between devices. Unlike Wi-Fi, Bluetooth comes with a firmware enclosed on hardware chips that prohibits any attempt of addressing its limitations on the edge.

In this dissertation, we propose Leveraging Resources on Anonymous Mobile Edge Nodes (*LAMEN*[1]), an edge computing framework to harvest and orchestrate edge resources for executing complex applications, like video analytics, that overwhelms the backend, while they remain infeasible on a single smart device. In addition to this, they use a communication protocol customized for edge computing to overcome the limitation of current communication standards and minimize the communication latency between edge devices. We envision this work to maximize the gain from edge data and to enable a new set of applications.

1.1 EDGE COMPUTING NETWORKS

1.1.1 WHAT IS EDGE COMPUTING?

Edge computing is a new paradigm for pushing applications' execution, away from the backend towards the edge of the network [18, 19, 20]. Edge computing prepares a wide variety of devices that does not belong to the network infrastructure for taking over bigger roles. A plethora of research has focused on utilizing hardware upgrades to increase intelligence at the edge, e.g., cellular base stations [21] and video streaming on APs [22]. This approach aims to increase the responsiveness of services and application to save on round trip communication latency and complicated management of cloud servers [23]. In this dissertation, we focus on end devices, one hop from users, and we call them *edge devices*. Some examples are smartphones, smart watches, security cameras, machine sensors, cars, and smart T.Vs. Each of these can use its resources to execute an applications, or part of an application and combine results, instead of relying on backend servers. For example, *video analytics* are

[1]This literally means, a magical pendant to fulfill wishes of its owner as in science fiction movies and novels [17]

offloaded to cloud servers with large computation power despite the fact of having an equivalent power distributed among devices at the edge that are waiting to be harnessed.

1.1.2 WHY EDGE COMPUTING?

Edge computing is believed to be the brain behind IoT and smartphones [24]. It is all about pushing processing closer to their data sources in order to solve four main problems: communication, latency, cloud management, and data privacy [20, 25]. *Communication* avoids the overhead to communicate and to maintain large amounts of data that results in traffic bottlenecks. *Latency* provides low latency responses to applications that can hardly tolerate the round trip communication to data centers. In *Cloud Management*, data centers and cloud servers requires high support costs for continuous maintenance and upgrades, while edge devices are low cost devices that can be easily handled. In *Data Privacy*, users are not comfortable sharing their private data with the cloud, and assuming that this would reveal their privacy, track their locations, or expose them to unexpected breaches. Thus, we propose pushing computations to the edge, rather than bringing data to the backend.

In short, we propose the creation of a virtually homogeneous larger set of resources out of the multiple heterogeneous resources sitting across multiple devices. This computational unit should be large enough to compete with the cloud for hosting a specific application whose resource requirements are beyond the capabilities of a single device.

1.1.3 COMMUNICATION ON THE EDGE

The introduction of Bluetooth v1.0 in 1999 witnessed a wide community acceptance, despite some limitation. Bluetooth followed up with multiple versions that enhanced power consumption, device interaction, and efficient packet design for bandwidth optimization [26]. Unlike other technologies, such as ZigBee [27], Bluetooth was adopted by hardware manufacturers and was included in handheld devices. Its community acceptance and hardware availability made Bluetooth the de-facto communication protocol for IoT and a strong candidate for edge computing. Despite enhancements over versions, Bluetooth still carries four main limitations low transfer rate, high latency device discovery, limited range, and interference [28]. Unlike other communication standards such as Wi-Fi, Bluetooth limitations are unaddressable by

the research community, since its firmware is locked on hardware chips. Addressing these limitations and customizing Bluetooth to work on the edge, will require an open source Bluetooth stack. Hence, as a component of *LAMEN* we present *BOSS*, a Bluetooth Open Source Stack, which will provide Bluetooth core specifications to enable research and customization per edge applications' need. Further, we present *BLINK*, a customized Bluetooth version which will enhance the rate of data exchange within a piconet, along with a case study that shows how edge computing can benefit from a tailored communication protocol.

1.2 USE CASES

In this section, we present several use cases where *LAMEN* is needed. However, *LAMEN* is not limited to these, it can be used to orchestrate the edge for hosting any kind of scenario involving resources or data on the edge of the network.

1.2.1 COMPUTATIONAL ALLIANCE

Computational resources, the controller of all resources, are mandatory for the execution of any application. In this section, we explain scenarios that requires computations above the capabilities of a single device and we show how *LAMEN* provides a better alternative.

Speech Recognition is a complex application that researchers have been addressing for long time. Whenever a voice query is available, it is compared to a database of commands using speech recognition algorithms; this requires heavy computations above the capabilities of state-of-the-art phones [11]. Hence, commercial products can either limit their supported commands to a few that can be done locally, e.g., Garmen GPS [29], or can upload a recorded speech query for over the cloud processing, e.g., Apple Siri [10] and Google Now [30]. The latter option was estimated to require 100x the computational resources compared to regular queries, e.g., looking up a contact. Therefore, *LAMEN* supports speech recognition applications by providing on demand resources from multiple devices in the proximity and synthesizing them towards a single powerful computational unit.

Augmented Reality is getting plenty of momentum nowadays for applications like online gaming. People have shown a huge interest in kits like Oculus Rift that enables them in step into the game [31]. However, augmented reality requires continuous connection with a PC to offload heavy computations. In *LAMEN*, we propose

to provide such computational resources through proximal devices that grant gamers the freedom to walk around, not just stick to a chair close to their PCs. We envision that this will open the horizon for developing a newer category of games that enables player mobility.

Video Analytics target real time operations, e.g., face detection, while the video is taken. This feature is currently not possible on single end devices; hence, the data is uploaded to the cloud. Just as it does in previous applications, *LAMEN* steps into the picture to prepare the medium to host computational tasks beyond the capabilities of its requesting device.

1.2.2 COLLABORATIVE SENSING

Collaborative sensing describes a spectrum of applications in which multiple sensors are required to detect an event. In cloud computing, every sensor transmits raw data to the cloud for processing. The main theme of an edge computing contribution in this application category is reducing the amount of raw data that is passed to the cloud, especially if it is redundant; instead, it passes the result of processing from multiple sources on a small scale that can be further extrapolated at the cloud.

Media Coverage

Assume that a media corporation, for example CNN, is rushing to cover an emergency situation in New York City. They dispatch their correspondents with coverage units to drive through the traffic. However, what might be a more convenient solution would be to ask pedestrians to shoot videos and upload them until the reporters get to the scene. However, although this solves the initial problem, but it might result in a large number of videos from personnel seeking CNN's reward. Not only does a problem appears in communicating a large number of videos, but there will also be a problem in processing them and selecting the best; this is another overwhelming task that could take time equivalent to what could be spent waiting for the reporters to arrive.

In edge computing systems, CNN would start by composing a resource request using a language that *LAMEN* provides to specify the incident location (Manhattan), the resources to be fetched (camera), the period of operations (5mins video). CNN would then send the request for the edge devices to the corresponding location. These scenarios consists of the following stages in order: composing a request, dispatching

to edge devices, extracting request resources, resource lookup, and reporting them back.

Restaurant Search

Current restaurant lookup applications provide static information, like the restaurant's hours of operation, menus, and addresses none of which could provide contextual information like current waiting time, serving times, or noise level. Edge computing enables recruiting resources in a group of restaurants and using them to execute noise level detection applications or estimating waiting times and reporting them back as a parameter in restaurant selection. Here is an example, Alice and Bod are looking for a restaurant in Manhattan for their anniversary dinner. Using their preferred app, Yelp, they are able to view various restaurants' menus, pictures, and operation times. However, they are interested in knowing some information about the restaurants, right now, that no app provides, e.g., is this restaurant noisy? What is its average waiting time? Using edge computing the couple opens a similar app, chooses a target location, and requests the noise level feature. Unlike other approaches that question users, like CrowdDB [32], which does not scale, edge computing provides an accurate feedback by processing data at the network's edge.

1.2.3 I/O VIRTUALIZATION

I/O devices are receiving their share of interest in smart device innovations, e.g., display resolution. However, the goal of keeping devices handheld limits both their size and their overall quality. Hence, to get a larger display or a deeper sound, multiple end devices have to be synchronized. *3D Story Teller* is a sample application that splits a story by character, each played on a different device [33, 34] as in this example: During kindergarten story time, teacher Linda opens *3D Story Teller* instead of reading the children a story on her own. The application searched the surrounding for collaborating devices and establishes a synchronized connection. Then, Linda's phone splits the story characters among the available phones, i.e., 1 to n, where n is the number of characters, and dispatches them to corresponding phones. *3D Story Teller* tackles challenges on the edge to provide a movie theater-like experience for the story audience.

1.3 CHALLENGES ON THE EDGE

Edge computing systems face a set of challenges: discovery, mobility, hardware heterogeneity, service management, rewards, and communication protocol. *Resource Discovery*, with plenty of resources in the proximity, is the discovery process required to discover and allocate suitable ones per incoming application. *Mobility*, unlike data centers, edge nodes, are always moving which contradicts the need for stationary resources. In *Heterogeneity*, users have different device preferences, e.g., iPhone, Fitbit, or Android, and such variations have to be seamless while executing edge computing applications. *Service Management* has multiple challenges to handle a service definition, recruiting a workforce, service dissemination, service-splitting among devices, execution monitoring, collecting, and aggregating results from multiple devices. *Rewards*, which is the main incentive for user participation, have to incorporate factors like the number of resources used, their scarceness level in the region, and responsiveness. Moreover, the reward negotiation has to reach a fair deal between the payer and the payee. A *Communication Protocol*, is needed as a prerequisite for providing the aforementioned features. The IoT community has looked at multiple communication protocols and has reached a consensus that Bluetooth v4.0 is the most suitable. However, with *LAMEN*, Bluetooth imposes a higher latency than what edge applications can afford, which is caused by slow discovery, unicast communication, and a low data rate.

1.4 *LAMEN*

In this dissertation, we present the design and implementation of *LAMEN*, a three-tier framework to orchestrate anonymous devices in the proximity and prepare them for hosting complex services currently performed over the cloud. *LAMEN* is unlike existing solutions that use edge devices to execute individual applications or to filter raw data before transmitting it to the cloud for processing. *LAMEN* hides multiple device heterogeneity, synchronizes them, and uses their collective resources towards a single application. Following is a the summary of *LAMEN* contributions:

1. We design *LAMEN*, a three-tier framework, to incorporate edge devices and to prepare them to host complex applications. In *LAMEN*, we explain the modules required to organize and harvest underutilized resources on participating devices.

2. We design and implement *Kinaara*, a framework that discovers and allocates resources with suitable QoS features in order to execute applications. *Kinaara* ensures resource discovery, upon availability, by exploring a maximum of O(logN) of the available devices. Moreover, *Kinaara* hides the identity of participants and keeps their private data on their own devices.

3. We design *DiScript*, a scripting language to compose edge application requests and to describe their detailed resource requests. *DiScript*, defines QoS metrics for each resources to be used, while negotiating resource recruitment.

4. We design and implement *DiCoder*, a customized resource encoding scheme to represent resource available in the system. *DiCoder* is used to express available devices in terms of their resources. This enables *Kinaara* to efficiently locate them whenever a service request is received.

5. Resource Recruitment following on the discovery results, the recruitment modules select the best subset to carry the incoming service.

6. Bluetooth Open Source Stack we design and implement an open source Bluetooth stack (*BOSS*). This module enables researchers to study and enhance Bluetooth's core components. Moreover, we present *BLINK*, a case study that addresses a limitation in Bluetooth, i.e., multicast communication in piconets, and shows how *BOSS* enhances the level of the data exchange.

7. We evaluate *LAMEN* using different metrics such as mobility, large network size, limited resource availability, and request rates. Then, we record the performance in terms of the accuracy of fulfilling a request, the number of hops, the latency, and the limits of scalability.

8. We implement three applications to show the behavior and the benefits of *LAMEN* enabled applications on the edge of the network.

1.5 ORGANIZATION

In Chapter 2, we discuss the background of cloud computing and edge computing systems. We also discuss attempts to manage and recruit a workforce in similar systems, i.e., sensor networks and P2P, as well as practical approaches to solve hardware

heterogeneity. In Chapter 3, we describe *LAMEN*, a framework used to incorporate edge devices in one system to enable complex application execution beyond the capabilities of a single device. In Chapter 4, we design, implement and evaluate *Kinaara*, a solution to index edge devices, and we locate suitable ones to serve every incoming service request, based on resource features and availability. In Chapter 5, we present *BOSS*, an open source Bluetooth stack for researchers to study and enhance Bluetooth core specifications. In Chapter 6, we present three applications and show the value of *LAMEN* contributes to multiple application categories on the edge of the network. Finally, in Appendix A, we end with a discussion of the factors and methods to incentivize users towards using *LAMEN*.

CHAPTER 2

BACKGROUND

Attempts to push computation to the edge have been happening for a while. Backed up by various motivations like utilizing the co-existence of multiple resources [35], decreasing the overhead of data communication [36], offloading [37], or privacy concerns about keeping personal data close to the user rather than posting it over the cloud [38]. Despite various motivations, researchers agree on the eligibility of edge devices for performing tasks beyond a single user's commands. However, their main objective was collecting raw data or performing computations on static networking components, e.g., AP, that doesn't support scenarios for mobile devices.

2.1 CLOUD, CLOUDLETS, AND FOG

Cloud Computing is a model that grants users access to shared computational resources, e.g., networks, servers, and storage, which are tailored towards a specific application. Cloud systems consists of a large number of resources, which are not affordable by regular users or by small business organizations [39, 40]. Commercial cloud services offer the flexibility to define resource needs along with date/time requirements for a pay-on-use only systems, which is cost efficient. Such services are offered by many providers like, Amazon Web Services [41] and theGoogle Cloud Platform [42].

The rise of cloud computing has granted access to the huge infrastructure on the network's backend servers, has provided a quick recovery from disastrous scenarios, has guaranteed security through encryption, and has reduced carbon emissions at the small business sites by 30% [43, 44]. However, it requires continuous Internet connectivity, ongoing payments to cloud providers, and privacy concerns as to where businesses should store their data [45, 46, 47, 48]. Hence, researchers are continuously addressing such limitations as they appear in real-life scenarios.

Cloudlet is an architecture that provides a local version of cloud computing as a virtual machines (VM) instantiated on a local server with LAN communication [49, 50, 51, 52]. This targets IoT devices to collect location-specific sensor

readings from large number of highly mobile nodes [53, 54, 55]. Another example is *smart traffic lights*, where light switching is based on the current vehicle's pattern collected by nearby sensors [52]. Fog Computing is another term that represents the same concepts and challenges as cloudlets.

Cloudlets enable the organization of the network's edge into small co-located groups, each of which acts as a local cloud. Since their inception, researchers have been eager to study edge networks in order to define their lowest level building blocks, their challenges, and what applications the edge can contribute to [56, 57, 58]. Much of their work has agreed on the same concepts under different names: *Task management*, to maintain information about tasks running at the edge; *Device management*, to keep updated information about the status of participating devices [59, 60]; and *Scheduling*, to synchronize the execution of multiple devices [61, 62]. However, discovering devices and resources has been achieved through third party solutions like Amazon Mechanical Turk (AMK) [63].

Although both Cloudlets and Fog applications targets target the formation of small cloud networks, each group work alone. They are only concerned about smart data collection. This is unlike *LAMEN*, which targets the orchestration of resources available in the near proximity to serve a specific application, then dissolves the orchestra. In other words, in *LAMEN*, whenever a task is available, a coalition is built to serve it.

2.2 CROWDSOURCING AND CROWDSENSING

Crowdsourcing is recruiting a short term workforce without any rigid employment obligations [64, 65]. Crowdsensing is using crowdsourcing devices with sensing capabilities [66]. The evolution of mobile environments has triggered *mobile crowdsensing* to collect sensing information from mobile devices [67].

Hence, Crowdsensing has been proposed as an approach for accessing sensing data on edge devices. This collective sensing is used to reducing power consumption by eliminating redundant sensing [68], creating video maps based on GPS locations [69], and combining various sensors from different devices for a single task. For example, RIO, enables Skype calls with camera, display, and microphone from different devices [70] and CrowdDB asks questions to the crowd [32]. In addition, applications targeting computational resources like Medusa [71], which recruits a set of devices, select a routine from pre-installed ones to execute, and get the results back. They

all share negotiations with a set of registered physical users to take the task. They maintain users, along with their device information in a database either through an application specific system or AMK [66]. This assumption is acceptable for such applications, but ones that need for low latency resource discovery can not afford the waiting time until users receive a message and respond.

It is worth mentioning that some approaches perform predictive models for user's location [72]; this is completely different from the kind of resource/device discovery that depends on the resource's status as the search criteria, not on a specific user. Also, in crowdsensing the status of devices is not a parameter in the selection. Hence, we need an approach to discover resources based on their up-to-date status (e.g. value, and availability) and to assign services in a low latency fashion. We envision *LAMEN*, if used with crowdsensing applications to make them smarter through the selection of the right candidates.

2.3 SENSOR NETWORKS

Similar to Edge networks, Sensor networks operate through nodes, i.e., motes, at the edge of the network. These networks consist of small size, cheap, and low powered motes with limited sensing and computation capabilities [73]. In Sensor Networks, nodes are dispatched in large numbers to cover a certain location, mainly for monitoring purposes [74, 75]. In a location, sensors are all programmed to perform a certain task at dispatch time [76]. Efforts to reprogram sensors in run time, i.e., re-tasking, using customized languages, e.g., SNQL [77] or TinyDB [78], include the massive selection of available nodes based on a criteria, for example, the amount of remaining power, and does not provide selective recruitment towards a specific application [79]. This would result in the over-recruitment of devices, which would not all be used and would blocks other applications from allocating idle resources.

Due to their limited computational capabilities, sensor networks are not suitable for hosting edge computing applications. Also, since they lack the wide variety of sensors per node, sensor networks are less intrusive to human life, when compared to smart devices, e.g., smartphones and IoTs. Hence, they areless likely to be available on the spot for an edge application in need. However, they have similar management requirements for their devices [80, 81]. Self-managing nodes have been addressed in electing leaders [82], choosing proper sleep/wake up intervals for

power constraints [83, 84, 76], mobility [85], resource discovery [86], and data aggregation [87].

2.4 P2P NETWORKS

Another domain for of using edge devices is the Peer-to-Peer (P2P) network, a distributed content sharing network in which all nodes are equivalent, i.e., peers, in terms of their functionalities and the tasks they hold [88, 89]. P2P's most prominent application is file sharing, e.g., Napster and Gnutella [90], which connects a large number of peers each carrying a file represented by a key/value pair. The key is usually the hash of the file name, while the value is the device's IP. Anyone who wants to find a file hashes its name and looks for the corresponding IP in a hash table. For scalability issues, centralized hash tables are proven to be inefficient solutions [91, 92]. Hence, Distributed Hash Table (DHT) approaches were introduced to split the hash table among all of the participating nodes. The hash table splitting and key generation hash function differ across multiple DHT solutions. The most famous solutions are Chord [93], Pastery [94], Tapestry [95], and CAN [96]. They all share the same concept of hashing the file name to represent the device where it is stored. Despite their efficient addressing, DHTs in this form are not suitable for dynamic applications, as their basic operations, i.e., node join, leave, and lookup, require $O(\log_2 n)$ time complexity, where n is the number of nodes in the network. Also, the naming convention does not reflect the node capabilities. PHT [97] and Hamming-DHT [98], proposed sorting P2P files based on the key similarity. Mapping their approaches to edge computing networks the former resulted in $O(n^2)$ operations, while the latter used an ambiguous key generation process when mapping the multiple resources of a single device. In general, P2P addressing was designed to map a single file (single resource), and falls short when representing a multiple of them in one key on the same device. Finally, ZHT [1], a highly optimized implementation for DHT applications, was evaluated on IBM Blue Gene with 160K-cores, and the Linux cluster with 512-core to result in very low latency operations, i.e., 1.1ms.

All of the aforementioned P2P solutions were built to reach a file stored on a stationary device, unlike edge computing systems where devices are mobile, and have plenty of resources not a single file with one feature, i.e., name. Also, P2P systems do not face the problem of heterogeneous devices. Therefore, if *LAMEN* is to support edge systems, it needs to consider mobility, resource status variations,

and heterogeneity which were not addressed in the P2P domain.

2.5 EDGE COMPUTING

Edge computing targets performing computations on devices close to end users and away from the backend [99, 90]. The recent evolution of smartphones and IoTs facilitates the bringing of Edge computing framework to life. However, these devices were originally designed to serve a single user. In the rest of this section, we will present attempts to use edge devices.

2.5.1 EDGE COMPUTING MIDDLEWARE

Middleware design is needed to support mobile device collaboration. *MECA* proposed an initial middleware architecture to bring raw data analytics to the network's edge. However, the architecture was very abstract and was only used for selecting data sources [59].

Medusa is a programmable framework to facilitate crowdsensing. It adopts the following five-step scenario for Alice to collect videos of a specific incident: she recruits volunteers to receive tasks, e.g., shoot videos, via sms or mms; they shoot and upload summaries; a reviewer picks the best summaries, and asks for the uploading of their full videos. To achieve this scenario, Medusa requires the service requester to write an XML, called a MedScript, to describe the subtask components and their order and the feeds it to the Medusa system. Medusa consists of the following modules: *MedScript Interpreter*, which analyze the input XML and makes sure that the sequence of components is correct, e.g., shoot video then extract summary; *Task Tracker*, which makes sure that tasks didn't exceed the time limit, decides the number of workers needed, and the monitors task; *Worker Manager*, which connects with Amazon Mechanical Turk to recruit volunteers, and assign incentives; *Stage Library*, which maintains code snippets required to execute system stages, e.g., face detection and data upload; *Stage Tracker*, which maintains user privacy and supports multiple stage execution; and *MedusaBox*, which makes sure that the tasks does not use more resources than the worker specified and maintains the worker's current state, in case of a limited time outage. Assuming that smartphones are dump, they only use it to perform the latter two modules, leaving all of the management and smartness to the cloud. It also requires continuous user involvement in approving every stage of execution. For example, to execute a code, the user receives an sms with a link to

download a video recording and summary extraction app. Then, the user receives another sms with a link to manually upload the full version of the video. Also, the XML creation by the service requester is not a user-friendly process, in fact it is rather primitive, if a user needs to capture a video, he or she will complete steps 1, 2, and 3, so there is no need for users to specify it every time. [71].

2.5.2 USING THE EDGE

CoMon establishes a Bluetooth connection between nearby devices; then, it performs sensing on a single device and broadcasts the results to others, to reduce the overall power consumption [68].

MiroBlog is used to build a location based on a map of videos. It collects videos from multiple sources, detects their location, and then displays a pin on every map location they have a video for. MicroBlog depends on users sharing videos to a cloud server through their cellular connectivity. It does not support direct collaboration between devices. However, the whole task to be completed requires the participation of many devices [69].

RIO is an Android system for virtualizing I/O resources between devices. For example, Skype video calls requires a microphone, a camera, and a display. RIO can use these three devices, with each providing a resource to perform the Skype call. RIO receives messages from multiple devices and writes them in a kernel file that is accessed by the currently running app. However, this approach enables only one service to be running at a time, which raises a scalability concern [70].

GameOn enables users in proximity to rely on their Wi-Fi Direct connectivity instead of on cellular for multi-player gaming scenarios. It provides wrapper functions for every game, to map the prospective P2P communication to cellular incoming data formats. This approach avoids any modifications in gaming apps, but requires the development of a wrapper function for every game the system supports. GameOn provides device-to-device coordination, to save time and the cost of using cellular service. However, the system lacks real collaboration between devices [36].

CrowdDB provides a scheme to crowdsource the results of customized SQL queries. CrowdDB provides a modified version of SQL which enables new fields or tables to carry a specific value, i.e., CNULL, stating this field or table is retrieved through crowdsourcing. Similar to Medusa, CrowdDB depends on AMT to recruit users to assign incentives. However, it provides the set of users with a customized UI

to provide results, e.g., *select url from university*, and compares the received results over the cloud to select the most repeated answer as the correct one. This approach is also used to map record values, for example, by IBM and International Business Machines, or to locate the shot for a specific picture. *CrowdDB's* dependency on AMT for user recruitment and incentive is a drawback. Also, the DB nature of Crowded makes it suitable for answering questions, labeling pictures, and performing simple tasks that require human feedback. It is clearly not suitable for tasks that include data processing on mobile devices [32].

Paradrop used the definition of static edge to reprogram access point with light weight code snippets [100]. The static edge refers to network infrastructure components that are close to the users, e.g., AP. In the work, the authors used a reprogrammable AP to install home automation application. The work used a case study to monitor the differences on room temperature and try to keep all rooms in the house at the same temperature. This work as others whose focus is the static edge limits their scope to applications that depends on an infrastructure leaving them with mostly indoor scenarios.

MAUI divides program execution between mobile phones and cloud servers based on annotations enforced in the code. More specifically MAUI maintains two copies of the program code, one running on the server and the other on the phone. Upon assessing the current situation in terms of energy consumption and networking, MAUI decides where to execute each code snippet [37].

Code-In-The-Air (CITA) provides location-based actions, it detects the user's current location, then performs an action accordingly; some examples include changing the phone ringing mode when entering a meeting room or notifying the user when a friend is close by. CITA has three main components: the *Tasking framework*, which allows writing scripts, compiles them into server/mobile codes, and manages task execution; the *Activity Layer*, which provides task abstraction through extensible modules to recognize a number of activities, e.g., isWalking, isDriving, and enterPlace, through old approaches, and then acts accordingly; *Push communication*, in which CITA claims high overhead in performing TCP sessions to inform the mobile side code to execute. Hence, they maintain a DB of phone numbers, each mapped to a service on the phone, and call the phone using one of the numbers. If a CITA client detects a call from a pre-saved number, it rejects the call and initiates the corresponding response [58]

2.5.3 MISSING ON THE EDGE

In this section, we presented a breadth of how the edge is currently used. However, none of the previous approaches have been concerned with managing the edge to execute applications. Most of the work has been concerned with supporting a specific type of applications for a narrow purpose, e.g., power consumption. Also, addressing is targeting devices regardless of the resources they carry, or what is their current status? This leads to recruiting unsuitable candidates. Further, users decide whether or not to accept an application, which imposes the latency of waiting for user's response. It is also worth mentioning that previous research has considered the edge as a static networking entity close to the user, e.g., AP, which has eliminated a wide spectrum of mobile devices with both a large resource set and technical readiness to be part of the edge.

2.6 COMMERCIAL SOLUTIONS

Commercial solutions provide D2D communication between devices within a certain proximity. They offer open APIs for easily adoption in industrial products, some of which are now in the market. In *LAMEN*, we adapt some of these solutions to provide D2D interaction between the heterogeneous devices in our testbed.

2.6.1 ALLJOYN

AllJoyn is an open source software framework, makes it easy for devices and apps to discover each other and to communicate with each other. Developers can write applications for interoperability regardless of transport layer, or the manufacturer, and without the need for Internet access. The software has been and will continue to be openly available for developers to download, and it runs on popular platforms such as Linux, Android, iOS, and Windows, including many other lightweight real-time operating systems [101].

AllJoyn is supported by the AllSeen Alliance lead by Qualcomm [102]. The alliance has over 200 members including Microsoft, L.G., Phillips, Sony, and IBM. In 2015, they introduced products that supports AllJoyn like the Panasonic Wireless Speaker [103] and LinkSys Routers [104], and others can be found on the alliance's documentation [105]. These products are ready to support intercommunication through AllJoyn messages and could be programmed to exchange information

and execute services.

2.6.2 IOTIVITY

IoTivity, is the AllJoyn counterpart adopted by Intel, Cisco, and Samsung. Similar to AllJoyn, IoTivity sends IP-multicast requests over LAN and receives unicast responses. It follows the same concept as AllJoyn, but it has a simpler implementation. Nodes receiving the multicast requests decide whether to accept by sending a unicast response to the source, or to reject by remaining silent [106].

2.7 BLUETOOTH FOR THE EDGE

Bluetooth operates in the 2.4GHz spectrum by splitting it into 79 channels, each with 1MHz bandwidth. In a single connection, Bluetooth guides the communicating parties, i.e., master and slave, to hop over the same channels each period of time in a process called *Frequency Hopping*. This allows devices to exchange information on a different channel each hopping time interval then move to the next and keep doing it till they decide to end the connection [107]. Despite enhancement over different versions, Bluetooth still carries four main limitations low transfer rate, high latency device discovery, limited range, and interference [28]. The problem, for the research community, with addressing such limitations, is that the firmware is locked on hardware chips. In other words, manufacturers have exclusive access to the firmware they develop for their chips. It is not expected that any manufacturer would either open his firmware implementation or address problems beyond the limited scope of his applications. We envision that the pre-requisite for addressing Bluetooth limitation is having an open source Bluetooth stack.

The Bluetooth stack consists of two components: host and controller. There has been some attempts made towards providing each of them alone. The Bluetooth host operates in the kernel space to provide high level features that are not tied to the Bluetooth core specifications. For example, it provides image profiles or device profiles. The host communicates with lower layers through APIs and use their data for high level features. Multiple implementations have been offered, each with limited scope. For example, BlueZ introduced the Bluetooth standard to the Linux kernel to interact with Bluetooth hardware chips [108], BlueDroid was adopted by Android OS to implement mobile specific features for a group of hardware chips [109], and BlueSoleil is a Windows-based application to enable interaction with other Bluetooth

devices [110]. These attempts were kernel-level implementations providing high level features, none of which included firmware or addressed core Bluetooth limitations, e.g., frequency hopping.

Bluetooth controllers, or firmware, are the main component missing towards an open source Bluetooth stack. There are some available development kits, however, these remain either closed-source and/or very limited. For example, Texas Instruments provide several Bluetooth development kits; such as PAN1323 [111] and EZ430-RF256x [112], as well as the corresponding Bluetooth Software Development Kit (SDK) such as EZ430-RF256x Bluetooth SDK GA [113]. Also, RetroPi is one of the few available open source kernels [114]. It is specific to raspberry pi with limited core Bluetooth features and support. However, the tools provided and the software does not grant developers any flexibility or access to the lower layers of the Bluetooth stack, such as the Bluetooth baseband layer.

Open source Bluetooth full stack is extremely scarce. To the best of our knowledge Mynewt is the only solution to offer a full open source attempt for Bluetooth [115]. Mynewt is an operating system for IoT devices that offers a Bluetooth open source stack with implementations for both host and controller. However, its master cannot act as a central device to manage multiple connections and piconets. Also, Mynewt is not compatible with BlueZ, since its implements its own host, which limits its interaction with other devices.

CHAPTER 3

MOBILE EDGE COMPUTING ARCHITECTURE AND COMPONENTS

3.1 SYSTEM OVERVIEW

In this chapter, we discuss *LAMEN*[1], an edge computing framework for orchestrating edge devices and using their resources in executing an application [116, 117]. We define an *application* as a code routine ready to run on end devices, e.g., noise level detection and health activity within a location. Each application requires a group of resources to operate on, e.g., sensors and a CPU; thus *LAMEN* needs to provide four main modules: resource lookup, resource orchestration, application delivery, and a rewards model. *Resource lookup* is used to discover and allocate the resources required by the application. *Resource Orchestration* is used to synchronize the performance of heterogeneous resources and present themselves to the application as a virtually single homogeneous resource. *Application Delivery* is an efficient way to deliver applications for end devices, as well as to monitor their performance, and to return their results. Finally, a *Reward Model* is used to compensate resources donors for their role. *LAMEN* locates and hires resources without revealing their owner's/device's identity or continuously tracking their location, i.e., GPS. In short, *LAMEN* prepares edge devices to host applications whose resource requirements exceeds what a single device could offer, in a way that is more efficient than going to the cloud.

Unlike centralized solutions, e.g., Micro-Blog [69], *LAMEN* proposes a layered architecture in which devices are clustered based on their proximity. Each cluster has a head node, i.e., a mediator, acting as the cluster coordinator and the contact to other layers. In *LAMEN*, we design mediators to keep track of devices and their resources within a cluster, to organize them according to their resource similarity, and then to recruit a group to execute incoming services. This design enables optimized

[1] A magical pendent to fulfill the wishes of its owner.

discovery for a group of resources, since they will be close to each other. Also, users can keep their data locally for privacy concerns. Moreover, resource owners maintain their identity anonymously at the mediator level, such that incoming services from different clusters have no clue who executed them, and know only the mediator who keeps the record to be used for incentives estimation.

3.2 DESIGN PRINCIPLES AND ASSUMPTIONS

The Edge is Dump

Despite the continuously increasing smartness of devices on the edge, they cannot benefit from their coexistence. Smartphones and IoTs are designed for personal use, although at a close proximity, e.g., university campus, the collective number and the quality of resources could easily exceed the capability of backend servers. We envision orchestrating the performance of these devices to result in high capabilities at the edge without the overhead of maintaining data centers. Hence, in *LAMEN*, we aim to change the nature of edge devices from being dump collaborators to being an active piece of the larger puzzle.

The Edge is Ready

Despite any dumbness at the edge, individual devices are getting smarter day by day. The previous few years have witnessed significant enhancements to serve their owners. We envision that edge devices in the market are ready to host edge computing if they are properly orchestrated. Hence, in all our upcoming design choices, we favor practical options despite others that could be theoretically better.

Dynamic Environment

In edge computing networks, devices in the user's custody are continuously on the move. This is unlikely to maintain any constant network topology, e.g., neighbor connections or APs, which contradict the goal of building a coalition of devices towards a service execution. Therefore, *LAMEN* has to seamlessly handle user mobility and guarantee resource availability throughout a service execution time.

Distributed Design

The dynamic nature at the edge does not fit centralized solutions, even though centralized systems requires easier management, guarantees accurate results, and generate quick responses. In such a dynamic environment replicas have to be maintained; this is not available on the edge. Thus, *LAMEN* should consider a distributed solution to locate resources, and to handle their service storage, and execution.

Resource Encapsulation

In *LAMEN*, we target harvesting edge resources for service execution. However, resources are not on their own rather, they are packaged on end devices, e.g., smartphones, laptops, or wearables. In this dissertation, we define a device[2] as a container of resources. Although our target is the resource, we will end up recruiting devices. Hence, *LAMEN* will consider optimizing the resource/device relationship in both recruitment and execution, according to the service requirements. In *LAMEN*, a device can be recruited by more than one application each using a different resource, while a resource, once recruited, is exclusive to a single application.

Maintain User's Privacy

Users are not comfortable revealing their identity, it is psychologically linked to compromising their privacy, either through location tracking or through resource usage patterns. Unless carefully handled, privacy concerns could discourage participation in *LAMEN*. This might seem to contradict maintaining a rating and history per user either for estimating rewards or for selecting historically reliable nodes. Hence, *LAMEN*'s mediator is designed to balance the trade-off between a user's anonymity and history tracking.

3.3 *LAMEN* EDGE COMPUTING

3.4 *LAMEN*'S LAYERED ARCHITECTURE

LAMEN, a three-tier architecture, consists of a cloud server, mediators and edge devices as shown in Fig. 1. *LAMEN* users interact using a mobile app to authenticate, to grant access to their device resources at their convenient time slots, and to

[2]We use the terms device, and node interchangeably

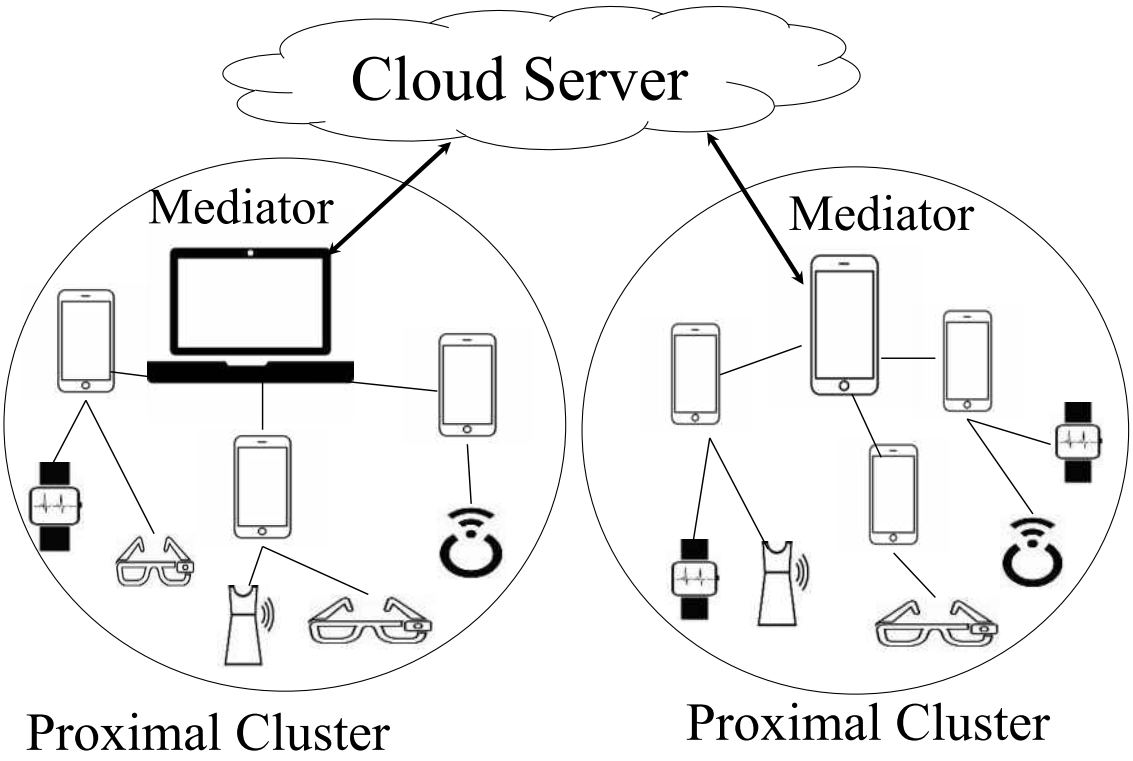


FIG. 1: *LAMEN*'s Edge Computing Architecture

issue resource requests.

Edge devices and proximal clusters. At the lowest layer are *edge devices* that opt-in to *LAMEN* to share their computing and sensing resources. Each *LAMEN* edge device may advertise local or non-IP connected resources. For example, a smartphone may advertise a gyroscope either as an on-board sensor or as a wearable sensor connected to it via Bluetooth. We say that devices sharing a certain context and a degree of trust among them form a *proximal cluster*, wherein devices communicate with each other either in a peer-to-peer fashion or via a gateway. For example, devices in a retail store, a restaurant, or a home form proximal clusters.

LAMEN supports multiple flavors of proximal clusters. The simplest proximal cluster can be a single wireless LAN in which devices communicate using wireless AP(s), Wi-Fi Direct, or Bluetooth. This is common for IoT scenarios and is currently being standardized, by AllJoyn [101], for example. Further, *LAMEN* envisions proximal clusters in which devices across multiple locations need to collaborate. For example, video surveillance cameras over a larger area may collaborate to infer that a theft is taking place, although such an inference may not be possible based on any single video stream.

Mediators are proximal cluster managers that play a key role in orchestrating cluster resources without the need for a cloud. Mediators have the following responsibilities: (a) establish a communication channel to the cloud server as well as to other mediators, (b) aggregate and expose the cluster resources to the cloud and to other mediators while protecting the identities of edge devices and their owners, (c) discover and register new devices and their resources in the cluster as well as detect their departure, and (d) respond to requests for resources. A mediator is a logical entity that can be physically hosted on any of the edge devices, e.g., laptop, AP, WLAN controller, or smartphone. Although a mediator can sit in an AP, its area of operation, i.e., cluster, spans beyond the AP limits. For example, in an enterprise, e.g., mall, covered by multiple APs a single mediator brings together all resources using an inter-AP communication, e.g., AllJoyn [101]. Further mediator-to-mediator interactions offer a wider variety of resources for edge applications.

If a mediator device leaves the cluster, a decentralized consensus algorithm elects a new mediator. Since the mediator election is expensive, the election algorithm takes into account the expected mobility of a candidate mediator and elects a device that has least expected mobility. Since such algorithms are well-known, this paper

does not provide details beyond using a k-leader election algorithm [118].

A cloud server is a repository for sensing and computing tasks (applications). It deploys them on proximal clusters and acts as a communication channel between proximal clusters. Also, it receives user requests for executing applications with known resource requirements. During resource discovery, it probes mediators for resources in their clusters. After resource discovery, it notifies the mediators to reserve resources and to send them the application code for execution.

Users. *LAMEN* engages three kinds of users: (a) device owners, (b) application users, and (c) application developers. Application developers submit resource requests to the cloud service, allowing them to develop and deploy applications without having direct access to the sensing and computing resources. The device owners opt-in to *LAMEN* by registering their devices and by specifying expected incentives. Smart contracts and Blockchain [119] can be readily applied for this purpose. Users acquire the application from the cloud server.

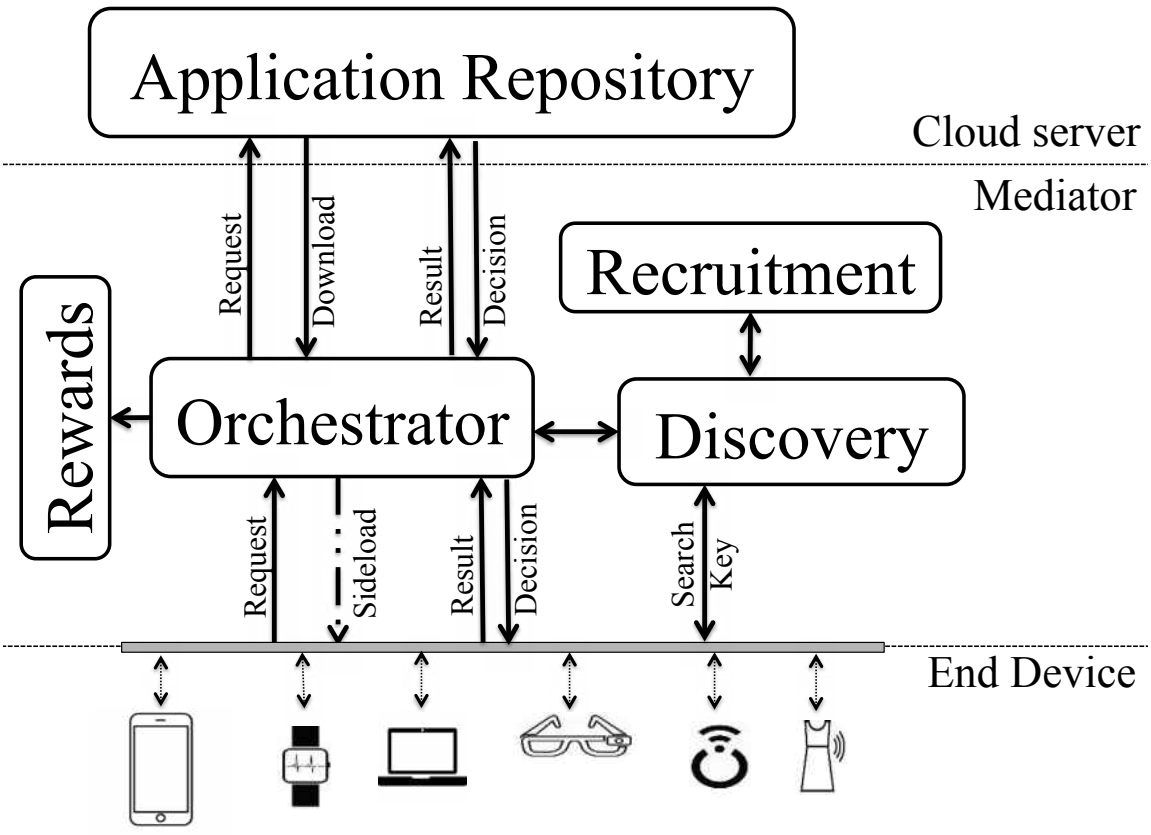


FIG. 2: *LAMEN* System Design

3.4.1 *LAMEN* SYSTEM DESIGN

In *LAMEN*, the *Cloud Server* maintains a set of applications, while *mediators* maintain information on available devices, select suitable ones per received service, deliver services to be executed, aggregate results, and finally reward devices for sharing. For that, *LAMEN* exempts *Mediator* from hosting applications in other words clears it from any job except managing the proximity.

In the case of multiple geographically collocated *mediators*, over-the-cloud communication between them might not be the best option. Hence, we propose the construction of a hierarchy of mediators between the cloud server and the edge devices to handle such situations. Each layer of mediators is managed in the same way as a group of proximal clusters. Details of such scenarios will be presented in Chapter 4.

3.4.2 APPLICATION HANDLING

Applications in *LAMEN* are code binaries ready for execution on the edge whenever *LAMEN* deploys them. Some examples include using a microphone to collect surrounding noise levels, using cellular connectivity to support uncovered devices, or even granting access to public pictures on devices for users who are looking for an incident. For those users, an edge application is designed to utilize resources previously offered by device owners. In *LAMEN*, any developer can prepare his application and can make it open for public use, in return for a reward. Details of what the application does are the developer's responsibility. However, *LAMEN* makes sure that the application can smoothly execute within an environment and that devices that can fulfill its prerequisites without compromising the user's security or privacy.

3.5 CLOUD SERVER

3.5.1 APPLICATION REPOSITORY

Cloud Server, the first *LAMEN* module shown in FIG. 2, facilitates communication between multiple proximal clusters, as shown in FIG.1. Also, it is an open layer for developers to upload their applications, so *LAMEN* users to request. To post an application in *LAMEN*, a developer has to register and upload his application in *Application Repository*.

Similar to Google and Apple App Stores, we coin the term *LAMEN Store* to describe the place that holds edge applications. Located over the cloud, the *LAMEN store* requires a developer's registration before posting their applications. Then, *LAMEN* users can select an application to be executed at their preferred location and time. The *store* carries the following features: it tracks service usage, maintains user rating per service, recommends services to users, estimates the price per service, and calculates the developer's reward.

Edge applications run on multiple heterogeneous devices and generate results in a different form. Therefore, each needs a different aggregation towards a unified decision or information. This takes us back to the question: is it a quiet restaurant? Thus, each developer has to include the corresponding aggregation service within his application.

3.6 MEDIATOR

The *Mediator*, the second layer, consists of the following modules: *Resource Discovery*, which locates a set of devices that hold the required resources; *Recruitment*, which selects suitable ones out of the previous set based on the service requirements; *Orchestrator*, which establishes communication between devices within a proximal cluster, sends services to recruited devices, monitors their performance, receives results from devices, aggregates them, and, if needed, reports back final results to the cloud; and *Rewards*, which receives resource utilization from the previous module, and uses it to estimate a reward to be used within the proximal cluster.

3.6.1 RESOURCE DISCOVERY

LAMEN's initial step in executing applications on the edge is locating resource requirements. Resources are scattered across multiple smart devices, e.g., smartphones, IoTs, or TVs. Thus, a resource discovery approach has to find devices that hold the proper resource, and has to make sure that they are available at the requested execution time. Moreover, resources of a kind have different QoS features. For example, twp devices may have cameras, but each have different frame rate, zoom level, resolution, etc. Hence, the *LAMEN* discovery approach has to provide a device with the right resources and availability to execute a service. To carry such discovery, first we must identify a procedure to compose a detailed resource request, post it on devices in the target proximity, and identify available ones to be used in

the next module: *Recruitment*. The resource discovery process is discussed in detail in Chapter 4

3.6.2 RECRUITMENT

Recruitment is extracting a subset of discovered devices to fulfill the request. In this section, we present the fundamentals of the *recruiter* module. Once a group of resources are discovered, the *mediator* should decide which nodes to appoint according to the following features: *Shortest Path*, *Expected Time Departure (ETD)*, and *History*.

Shortest Path

Shortest path selects the minimum number of devices needed to fulfill the resource request. For example, if the discovery request was looking for three camera, and three accelerometers, and one GPS, the discovery path might came back with five devices all with accelerometer, three with cameras, and three with GPS as follows: A: Accelerometer, GPS; B: Accelerometer, Camera, GPS; C: Accelerometer, Camera; D: Accelerometer, GPS; and E: Accelerometer, Camera. In this case the request could be handled only through devices *A, B,* and *E*. In case of an explicit mention that specific resources need to be on separate devices, the optimal allocation is replaced by the user's request.

Estimate Time Departure (ETD)

Every proximal cluster has to maintain track of its mobility pattern and estimates the average time spent by a device within its range. *LAMEN* through the recruitment module tracks that time and generates a continuously updated local average. Once a device joins a proximal cluster, the time is recorded and when it leaves, the time spent is calculated to update the local average. Therefore, when a group of devices is discovered that contains at least one device that spent more than or equal the local average it is discarded from the ongoing recruitment. Further, if the estimated execution time will exceed the local average, the device will be exempted from the current recruitment.

History

The *tracking history* is the cumulative ranking received per device upon executing a service. Seeking the best candidates, *LAMEN* maintains history of devices in terms of latency, service completion, and accuracy. *Latency* measures the time spent on execution, and studies whether the device usually performs other concurrent apps delaying the execution. *Service Completion* is the percentage of times that the device has to quit after accepting a service. Finally, *Accuracy* describes the way in which the returned result from a device is compared to peers in the proximal cluster, and the percentage of generating outliers. According to the previous three metrics, *LAMEN* generates a rank per device this rank is updated upon every service completion and is used as a parameter in recruiting it for the upcoming service.

3.6.3 ORCHESTRATOR

Orchestrator is the point of contact between end devices, and serves as their mediator. It is responsible for passing services to devices, and for receiving their results, as shown in FIG. 2. Moreover, it receives a request from proximal users, and determines whether it can be fulfilled within the current cluster using available services. If not, the request and its service are forwarded to the corresponding *mediator* through the *cloud server*. Then, the *orchestrator* can initiate discovery and can distribute the service among the recruited candidates. To achieve those responsibilities *Orchestrator* must provide three features: *Proximal Services*, *Sideloading*, and *Aggregation* using a set of *Control Messages*.

Control Messages

Upon establishing a communication channel between devices, *middleware* uses five types of messages to reach out to other layers: *Request*, which describes the to receiving of service requests from end users and the passing of them to the cloud server; *Download*, which describes bringing services from the cloud to the mediators; *Sideload*, which describes the passing of services between end devices under a single mediator; *Result*, which describes the collection of results from end devices and the passing of them to decision blocks; and finally *Decision*, which describes the passing of the information to the initial requesting device.

Proximal Applications

Proximal applications are used by mediators to specify the amount of applications he can hold with their last usage timestamp, given that all *LAMEN* users are within proximal clusters and that each of them can issue a request. When *orchestrator* receives a request, it checks whether the mediator is within the area of operation, whether the requested application is available within proximal list, and whether the available resources can perform the service. Otherwise, *cloud server* downloads the application to the right *mediator*. Once a new application is received it has to be placed in the proximal application list. Least Recent Used (LRU) is used for replacement in the case of a full list. Then, it is passed to the corresponding end devices.

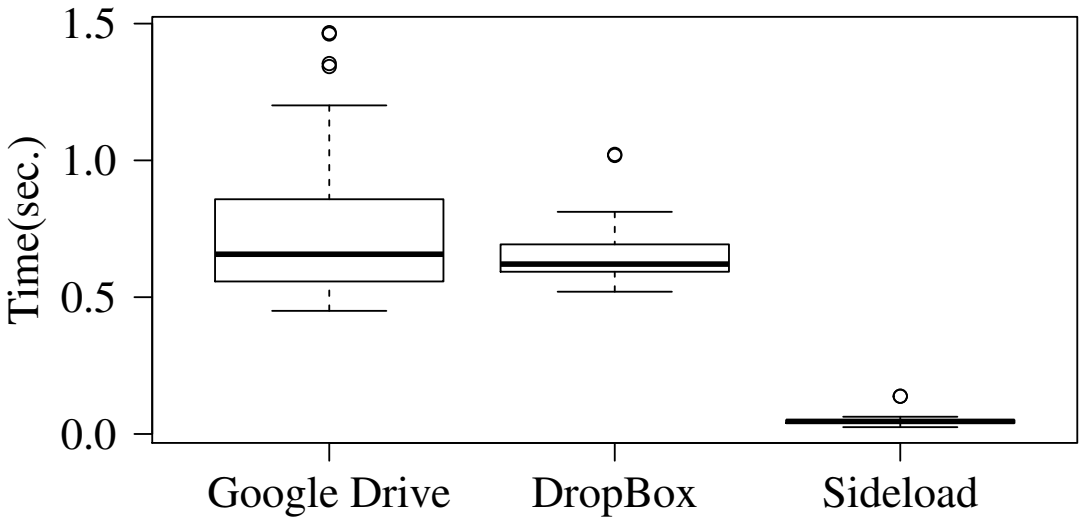


FIG. 3: Side Loading vs. Downloading from 2 different cloud providers 50 times each

Sideloader

Sideloading is downloading applications through local storage media, rather than through servers over the cloud [120]. In *LAMEN*, we choose to perform sideloading through the mediator, rather than to provide a URL for every device to download.

We envision that this will save the device's bandwidth and network usage, and will maintain privacy as we explained earlier, and as is shown in FIG.3.

Aggregator

Aggregators are used for the same reasons as the sideloader, but in the uplink. Results received from *recruited* devices are used as an input to the corresponding aggregator, which generates a single result to be passed to the *cloud server*. Finally, the cloud sends to the service requesting user through his *mediator*.

3.6.4 REWARDS

Unless they are rewarded, users will be reluctant to using *LAMEN*. Hence, we propose the use of a cooperative approach to reach a mutual and fair agreement between both the requesting user and the recruited devices. *LAMEN* cloud servers provide mediators with rates per resource. Mediators monitor the performance, calculate the resource usage per service, and distribute the reward over participating devices. So far, we have kept it simple; however, we will consider a more adaptive approach in our future work. For example, the rate for seizing an accelerometer available while having 10 units should be different from having a single unit within a cluster. Also, a user rated for good results should differ from others. Moreover, requesting users should be enabled to bid or to limit their payments.

3.7 EDGE DEVICES

LAMEN targets the utilization of resources on edge devices at the user's hold. Users willing to be part of *LAMEN* have to register their devices and install a *container app*. The app enables selecting resources to share, to select time sharing preference if any, and optionally to specify a value per resource usage. Later, the App works in the background to receive services and to set up their binaries to run, as well as to remove them after task completion.

3.8 HOW *LAMEN* WORKS

LAMEN consists of a collection of applications available over the cloud for registered users to request as shown in FIG. 2. Let's revisit the restaurant lookup

scenario as mentioned earlier in § 1.2. Users select the *noise level* application, identify the lookup region, and submit a *request* to the *cloud server*, which *downloads* the application to *mediators* in Manhattan.

Mediators parses the request, and generate a *search key* for devices with a microphone, an accelerometer, and a CPU available for sharing. Then, it initiates the *device discovery* module to match the *search key* with available devices, and the corresponding devices are passed on to the *recruitment* module. Checking their availability and features, a subset is selected and is passed to the *service middleware*, which sends the service to the recruited devices for execution, i.e., side loading. Each service operates on a device and retrieves the noise levels from the seated users (because others might be just walking around the restaurant and would not provide accurate readings). *Results* are averaged per mediator to express the proximity, and sent back to the *cloud*. Upon sorting them, Bob and Alice receives feedback in the form of list containing N restaurants, e.g., 5, sorted by their noise levels in Manhattan to pick accordingly. Meanwhile, devices in the selected restaurant will receive their compensation through coupons towards their orders.

3.9 MOBILITY MANAGEMENT

Since edge devices are mobile, it is common for them to move across proximal clusters, e.g., a device owner might be leaving the home cluster to join the work cluster. Mediators detect device departures using heartbeat and network timeout techniques. Similarly, the device itself detects a loss of connection to a cluster and starts discovering other clusters via standard service discovery protocols [121]. If a device leaves the cluster while executing an application, the *mediator* assigns the same application component to a different edge device. This transition may cause a loss of data and computation that *LAMEN* handles in a variety of ways depending on the application requirements. *Mediator* notifies the application of such events and the application handles such losses. For example, the loss of an application component processing a video stream requires no special handling given that the missed video frames.

CHAPTER 4

KINAARA: DISTRIBUTED DISCOVERY AND ALLOCATION OF MOBILE EDGE RESOURCES

4.1 OVERVIEW

Kinaara is a framework for discovering and allocating collective computing and sensing resources for edge computing applications [122, 123]. *Kinaara* has three fundamental design principles. First, it uses a multi-tier architecture to geographically organize proximal edge devices in clusters and provides their collective resources via trusted mediator entities. Second, it uses a novel resource representation to name and encode devices in terms of their user-advertised resources. Third, it uses a novel distributed indexing scheme to organize devices in a proximal cluster based on their resource similarity.

Kinaara possesses a variety of distinguishing features. First, *Kinaara* enables scalable and fast discovery of collective edge resources without revealing the identities of individual edge devices or exposing them to a public network.

Second, *Kinaara* enables a rich set of resource discovery primitives. *Kinaara* queries can identify multiple devices that satisfy a wide range of requirements particular to edge applications, including exact, aggregate and range query matches, multiple and heterogeneous resources and devices, and co-location constraints of resources on devices.

Third, the *Kinaara* indexing scheme's design, a ring logical structure, effectively handles device mobility and changes in resource status by encoding multiple resources and their features in a single key and by employing efficient basic ring operations (join/leave/lookup) that incur minimum overhead.

Fourth, *Kinaara* empowers edge application developers to express application resource requirements that go beyond single device boundaries. For example, in a video analytics application, the developer can express requirements such as the number of cameras and their resolutions for video acquisition and the collective CPU and memory requirements (which may exceed a single device's capabilities) required

for timely video processing. *Kinaara* also enables application users to specify application context such as location and execution duration. Based on user-provided context and application requirements, the *Kinaara* resource discovery and allocation mechanism can automatically identify a set of matching devices and can allocate resources for deploying the application components to these devices. During execution, the mediators act as aggregation points that disseminate results from edge devices to users.

We evaluate *Kinaara* using extensive simulations and mobility traces from a large public Wi-Fi Internet dataset with 150K users and 345 Wi-Fi Access Points (APs). The evaluations include *Kinaara* performance under different system parameters and mobility dynamics, and comparison with alternative resource discovery approaches. Our evaluation shows that *Kinaara* reduces discovery overhead by 70% and 40% compared to Chord [93] and that it offers a centralized scheme, and that performance degrades gracefully under high mobility.

4.2 BACKGROUND OF DISTRIBUTED DISCOVERY AND ADDRESSING

The vision of Edge Computing has been initially articulated through Cloudlets [49] and fog computing [55], which focus on the computing resources of the edge network infrastructure instead of on mobile devices. Recent work has proposed that using computation of clusters of mobile devices [124, 125]. Resource discovery has not been the focus of the Edge Computing work. Most work has been focused on problems such as migration [49], decreasing data communication overhead [36], computation offloading [37], scheduling [124], or fault tolerance [125].

Grid Computing systems have addressed distributed resource discovery and typically employed P2P overlays [126]. *P2P* systems are designed mainly for file sharing applications (also considered file discovery), which maps to resources in our case [93, 96]. The key space of such approaches is typically generated with hash functions and therefore devices are not grouped based on their resource similarity. In addition, P2P discovery approaches target exact match queries and, as we will show in the evaluation section, they are not efficient for searching collective and heterogeneous resources.

Resource discovery has recently been addressed in the context of the Internet of Things (IoT), which is a form of edge computing. Recent IoT standards, i.e.,

AllJoyn [101] and IoTivity [106], offer name-based discovery using IP multicasting, where a device either advertises its presence or identifies others. These approaches do not support a notion of single or collective resource discovery such as *Kinaara*, however, their open source is programmable. These approaches also operate under single WLAN scenarios, and scaling them to larger networks is challenging. Moreover, they require enhancements to execute low latency applications. The work in [127] proposed a global resource registry that would provide open APIs for search and resource matching. This is a centralized approach and has scalability issues. Furthermore, it targets returning the addresses of individual devices instead of using a mediator that exposes just their resources. This work offers a visionary approach without any evaluation metric. A recent work proposed a hierarchical architecture in which distributed gateways index devices connected to them and report them to the cloud [128]. This approach reaches to the gateway in a distributed fashion, while the gateway indexes the resources connected to it in a centralized manner. This approach introduces a single point of failure. It also supports simple exact match queries based on resource type and location and does not support complex queries for collective and heterogeneous device sensing and computation resources like *Kinaara*. DRAGON uses a tree structures to index resources and support range queries [129], and aims at indexing each resource separately to guarantee efficient tree search. Maintaining this large tree is not robust, as to mobility, and it requires efficient aggregation between multiple devices, which consume edge resources.

CrowdSensing is similar to *Kinaara*'s envisioned applications. It leverages the sensing capabilities of edge devices for multiple applications. Examples include reducing power consumption from redundant sensing [68] and I/O virtualization [70]. Discovery is either not addressed or is achieved through third party solutions, e.g., Amazon Mechanical Turk (AMK), which do not expose the resources per device [71]. Also, they require a human to be in the loop, which is different from than the applications envisioned by *Kinaara*, in which there are complex computation and sensing applications with real time requirements running on the edge and there is collective resource usage that may exceed the device boundaries.

In summary, although the above approaches were efficient in their domains, using them on *Kinaara* edge computing applications would result in communication overhead and would raise security concerns about exposing devices to public networks. In this work, we introduce mediators to eliminate redundant communication

and to hide the identities of collaborating devices. In addition, previous work does not efficiently support the scalable and efficient resource discovery of collective and heterogeneous resources with dynamic values and device mobility.

Earlier in Chapter 2 we explained prior attempts for using the edge. More specifically, we mentioned ways in which other systems using edge nodes have considered the issue of discovering (e.g. Crowdsensing, Sensor Networks). However, in this section we focus on Distributed Hash Tables (DHT) [130, 131], since they inspired our resource discovery solution *Kinaara*.

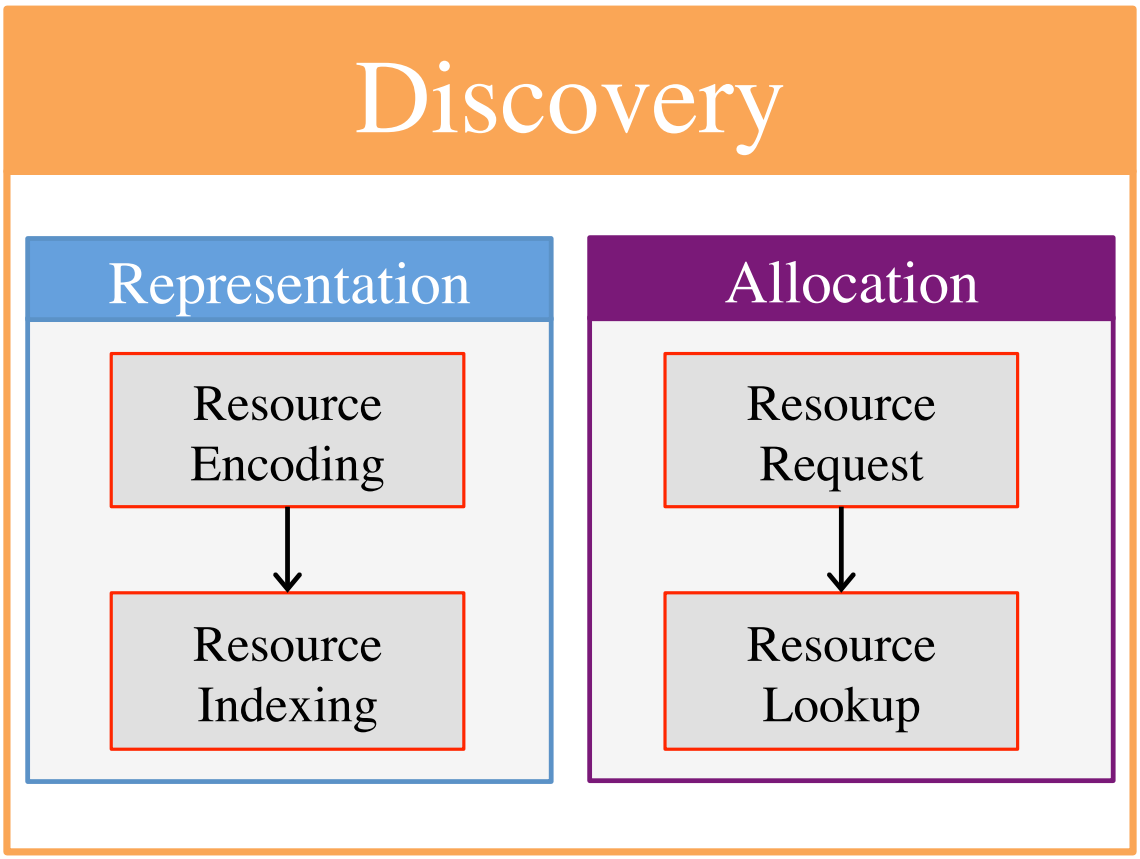


FIG. 4: *Kinaara* Phases of Resource Discovery

4.3 RESOURCE DISCOVERY

In the remainder of the paper, we will focus on resource discovery and its modules, as shown in FIG. 4.

Kinaara discovers resources at the network's edge based on their availability and their readiness to execute an application. As shown in FIG. 4, the discovery process

has two phases: resource representation and resource allocation. *Resource Representation* (§ 4.4) names and indexes devices within a proximal cluster; *Resource Allocation* (§ 4.5), analyzes resource requirements per incoming requests and discovers devices matching these requirements.

Throughout both phases, *Kinaara* uses the following components: a *key* to represent resources and their attributes for each edge device; a *Similarity Table* wherein each device holds pointers to other devices with similar resources; and a *Similarity Function* used to identify how similar two or more keys are to each other, e.g., via Hamming distance.

4.4 RESOURCE REPRESENTATION

Devices consist of a set of computing, memory, and sensing resources. Each is characterized by a set of features. For example, (a) a camera resource has a frame rate and resolution features, (b) an accelerometer resource has a sampling frequency and accuracy features. A key challenge in representing such resources is the sheer heterogeneity of the types of device resources and features. Another challenge is in enabling scalable search for suitable resources across all proximal clusters. In this section, we describe the novel techniques of resource encoding and resource indexing that enable *Kinaara*.

4.4.1 RESOURCE ENCODING

Discovery Encoder (Dicoder), pronounced as "Die Coder", is the module responsible for mapping resources on a device to a name, which will be called the *key*, from now on.

The main novelty of the *Kinaara* resource encoding scheme is to enable each *mediator* to encode a small subset of all the resource types and features. The cloud service maintains a dictionary of all the resource types and features recognized by *Kinaara*. The dictionary maps each resource type, feature types, and feature values to a unique binary code, e.g., $\{camera \rightarrow 1011000110011111\}$. When a *mediator* opts-in to *Kinaara*, the cloud service shares the current dictionary. Each *mediator* chooses the maximum number of resource types it encodes, for scalability, and comes up with its own local dictionary. For example, if a *mediator* chooses to support a maximum of eight resource types, it allocates three bits for the mapping of first eight resources it encounters in the proximal cluster. Thus, the domain of the local dictionary is

TABLE 1: Resource/Feature Encoding Guide in *Kinaara*

Resource			**Feature**	
Name		**Value**	**Name**	**Value**
Camera		001	Size	4 bits to express min. of 14 common picture sizes [132]
			Orientation	0 = not supported, 1 = supported
			Bit per pixel	4bits flag, with 4megapixels increment. up to 64 megapixels
			Frame rate	2 bits, with an increment of 10fps to support up to 40fps
			Focus	0 = not supported, 1 = supported
			Flash	0 = not supported, 1 = supported
			Snapshot	0=Pic.,1=Video
Network Status		010	Interface	00=Bluetooth, 01=Wi-Fi, 11=Cellular, 10=Wi-Fi Direct
Audio[133]		011	Frequency	2 bits, each increment supports 3 of the 9 common rates
			Channel	3 bit to map the 6 channels of 5.1 sound system
			Sampling Rate	2 bits, each increment supports 3 of the 9 common rates
GPS		100	—	—
Sensor	Magnetometer	101	Frequency	2 bits, with an increment of 10Hz to support up to 40Hz
	Accelerometer	110	Range	2 bit, each increment represents 5rad/sec
	Gyroscope	111	Direction	2bits to express the 3 axes

a much smaller subset of the global dictionary, which is dynamically constructed as *Kinaara* discovers edge devices. For example, assume that the first resource witnessed by the *mediator* is a camera, the local dictionary would be $\{camera \rightarrow 000\}$. When the *mediator* communicates its aggregated resources to a cloud service or to other *mediators*, it maps the local codes to the global ones. In the case of expanding the cluster-supported resources, the mediator informs all of the devices to modify their keys by padding 0s to the MSB per resource. The padded 0s double the resource bits to minimize any repetition of such low performance operation.

Similar to the resource encoding described above, each feature associated with a resource type is also encoded via a dictionary mapping. For simplicity, without loss of generality, we assume two feature types: *binary* and *range*. *Binary* features indicate whether or not the corresponding feature is available on the resource, e.g., a camera flash. *Range* features discretize a domain of values to a small number of categories wherein the domain itself can be discrete or continuous. For example, they can discretize a range of camera resolution features into "high", "medium", or "low".

FIG. 5: *Kinaara* Device Key

The codes for a resource and its associated features are combined into a *chunk* via their concatenation in a pre-defined order. Similarly, chunks corresponding to all resources offered by a device in the ascending order of resource codes are combined to form a *key* in the format shown in Fig.5. Note that *Kinaara* does not require the *key* to be unique, i.e., devices with identical resources will have identical *keys*. For example, device A with key "**001**0111**011**11" consists of two resources camera (code="001") and accelerometer (code="011") shown in bold. The camera supports a resolution of 8 Mpixel images (code="0111") and frame-rate 10 fps (code="1"), while the accelerometer reads with frequency 10 Hz (code="11"). Table 1 shows the encoding function used by *Kinaara*.

Finally, resource similarity plays a major role in the indexing strategy, as described later. *Kinaara* employs hamming distance between keys to estimate any similarity in the resources offered by two devices. If the *keys* correspond to devices

with no common resources, their distance is deemed infinite.

4.4.2 RESOURCE INDEXING

Kinaara indexes resources in a distributed fashion in order to avoid maintaining replicas for centralized indexing and to cope with the *mediator's* mobility. In this component, we explain how devices join and leave *Kinaara* using their keys. *Kinaara* creates a link between resource-similar devices through a structure we call *"ring"* to enable easy resource lookup. For example, a request for a camera from a device whose camera is currently busy is forwarded to another device with a similar camera instead of denying the request. In this work, similarity expresses how close resource features are between devices and is measured by the sum of Hamming distances between keys.

In *Kinaara*, resources are dynamic in *Magnitude* and in *Availability*. *Magnitude* represents the number of resources that an edge device selects to share at any time. *Availability* shows whether a resource is currently busy. Modifying the magnitude requires a new key generation, which then joins the ring as a new device. *Kinaara* detects availability during resource discovery. Our experiments have shown that representing the magnitude only in the key yields better performance. *Kinaara*'s indexes cluster through a ring supporting device as they join and leave operations.

***Kinaara*'s Ring**

Inside a proximal cluster, *Kinaara* sorts devices on a logical ring by their resource similarity. In order to maintain the ring structure each device carries a direct link to its successor and to its predecessor. Moreover, each device carries a data structure called a *Similarity Table*, whose entries are the most resource similar devices on the ring. This structure creates a link between similar resources on the ring enhancing the lookup operation. To mitigate an oversized *similarity table*, *mediators* limit similarity entries to logN of cluster devices.

During the initiation of the ring, *Kinaara* will have similarity gaps, in which non-similar devices are placed next to each other on the ring. The reason is that initially, devices with different resources may be joining the ring. Our assumption is that the number of devices will be much larger than the number of distinct resources within each device. After multiple devices join, the distance gaps between the devices will be smoothed out as more devices with similar resources join.

```
 1: Function deviceJoin (Node n, Key k, Node nearestNode){
 2:   IF (simFunction (n.key, k) <= 1)
 3:      appendNode(n,k)
 4:   else
 5:      IF( isEmpty(n.successor) )
 6:         appendNode(nearestNode, k)
 7:      else
 8:         nearestNode = nearestNode.closer(n)
 9:         deviceJoin(n.successor, k, nearestNode)
10:      end IF
11:   end IF
12: end Function
13:
14: Function appendNode(Node n, Key k)
15:      Node newN = new Node(k)
16:      newN.successor = n.successor
17:      newN.predecessor = n
18:      n.successor = newN
19:      (newN.successor).predecessor = newN
20:        newN.similarityTable.add("0000")
21:      Loop ( count(newN.simTable) < length(k))
22:         newN.simTable.add(newN.successor.simTable)
23:         IF ( count(newN.simTable) < length(k) )
24:            newN.simTable.add(newN.predecessor.simTable)
25:         end IF
26:      end Loop
27:      announcePresence(newNode)
28: end Function
29:
30: Function announcePresence(Node n)
31:      Loop i = 1:length(n.similarityTable)
32:        Node p = n.similarityTable(i)
33:        IF (simFunction(p.simEntriess, n.key) < simFunction (p.key, n.key))
34:           p.similarityEntry.updateSimTable(n.key)
35:        ELSE
36:           p.updateSimilarityTable(n.key)
37:        END IF
38:      END LOOP
39: END Function
```

FIG. 6: *Kinaara* Device Join Algorithm

Device Join

An edge device joining the ring sends the *mediator* resources that it is offering, along with time slots. The mediator generates a *key* and forwards the join request to the closest device in its *similarity table*. This is repeated until it finds the closest device, places the new device after, creates links to predecessor and successor, and builds the *similarity table*.

Throughout its search for the right location on the ring, *Kinaara* maintains a variable called "*nearestNode*", that is updated every hop between two edge devices if the current device is more similar than the previous. When the similarity function cannot find closer entries, it uses *nearestNode* as the new location. In case of duplicate keys, it places them as successors and the lookup can fetch their resources.

FIG. 6 depicts the details of device joining algorithm. After *Kinaara* finds the location for the new device on the ring (lines 2-9), it creates a two-way pointer with successor and predecessor (lines 15-19). Then it builds its *Similarity table*, and the first entry is always the mediator to maintain a direct link, i.e., "0000", (line 20). Then device's spatial locality, in terms of resources, enables borrowing the similarity table entries from the successor, in order to save the effort in having to look through the entire cluster. *Kinaara* applies the Hamming distance similarity check on the borrowed table entries, and the new device selects candidates for its similarity entries by iterating between predecessors and successors repeatedly until it fills its table entries (lines 21-26).

Moreover, the new device announces its presence to cluster devices (line 27) by reaching out to its similarity entries and their entries and contacts eligible devices to include the new devices in their tables (lines 33-37).

Device Leave

Detected by the successor and predecessor, once their direct link with the device drops, they establish a link together to mend the broken ring using a knowledge that every device keeps, i.e., successor of successor and predecessor of predecessor. Since the device which left is still in other similarity tables, *Kinaara* removes these links individually whenever any results in a communication failure. We chose this approach, rather than broadcasting a device leaving the event, in order to eliminate the overhead imposed by reaching out to the whole cluster.

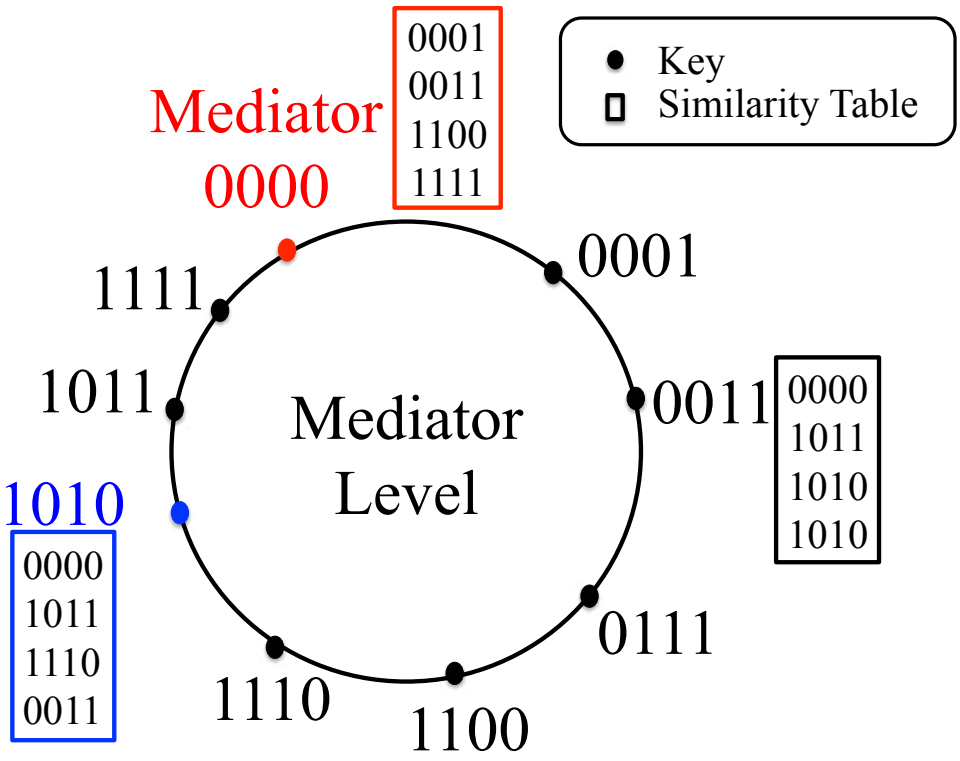


FIG. 7: *Kinaara* Sample Ring

4.4.3 INDEXING SCENARIO

FIG. 7 shows a sample ring in *Kinaara*. For simplicity each device holds a single resource represented by four bits chunk (resource and features), this can be extrapolated on the application's need. When the new device "1010" joins, the *mediator* searches his similarity table for the most similar device and forwards its control to its similarity table, i.e., "0011", which has a closer device, i.e., "1011". Hence, the new device request is forwarded to a new ring location and after checking its *similarity table* and the two following hops, it finds its location prior to 1011. Therefore, the new device is placed on the ring and readjusts pointers to "1110" and "1011" such that it is the successor of the former and the predecessor of the latter. Meanwhile, it points to "1110", and "1011" as its predecessor and successors respectively. Building a four-entry similarity table, the mediator comes first and the rest is borrowed from neighbors. Now, the ring is ready for resource requests.

4.4.4 MEDIATOR-TO-MEDIATOR INTERACTION

In an effort to incorporate the largest possible resources per application, *Kinaara* employs direct mediator interaction. This enables applications to fetch resources from the surrounding mediators either because of resource shortage or to expand the geographical area of operation.

Similar to devices in *Kinaara*, mediators follow the same procedures in establishing a ring, except for the key generation and the similarity table. In a mediator ring, *Kinaara* uses only available resources in the key generation excluding features. Our experiments showed that mobility caused high feature variation compared to resources, hence, they were ignored to generate a mediator key that would be less susceptible to changes. In the mediators ring, similarity tables holds two extra columns, *location* and *LRU*. The location holds a geographical location (i.e., latitude, longitude and elevation) that enables applications to expand their operations in specific locations. The LRU field holds the latest time of calling the corresponding mediator. Eliminating features from mediator keys generate higher redundancy; hence, LRU ensures a fair distribution of resources on applications.

The mediator's life cycle on the ring has three phases: join, modify, and leave. After establishing the device ring, a mediator can generate its key and use it to *join* in the mediators ring and create the corresponding similarity table. In the case of a

new resource showing up at a mediator, a new key is the generated and another round of mediator advertisement takes place to *modify* the previous key. When a mediator *leaves* its cluster, we follow the same lazy approach as the devices to discover the failure through failed communication (§ 4.4.2). Then, a new mediator is elected and joins the mediators ring.

4.5 RESOURCE ALLOCATION

On an established ring, *Kinaara* is ready to receive resource requests. To initiate resource allocation, *Kinaara* goes through two steps *resource request*, and *lookup*. On *mediators*, the former generates a request out of the application's required resources, while the latter performs the resource lookup.

TABLE 2: Search Query Prefix

Feature	**Value**
Count	4 bits, 4 devices per increment
Priority	1 bits showing optional or mandatory
Diversity	1 bit, "1" = resources on a single device

4.5.1 RESOURCE REQUEST

Mediators intercept incoming requests to create a resource query for discovery. The request is an application that includes a manifest file with detailed resource and feature requirements, like an android app's structure, which *Kinaara* uses to generate a *search query* with the same format as *keys* (§4.4.1), except for a prefix. Table 3 shows features of the prefix: *Count*, required amount per resource; *Priority*, whether the resource is mandatory or optional; and *Diversity*, which says whether resources must be discovered on the same device.

Range Queries target a range of features per resource. *Kinaara* codes its features in small ranges; if the query range is wider than the feature ranges, it is split among them, generating multiple requests corresponding to the main request's feature ranges.

```
<xml>
  <dscript>
    <user>WillSmith</user>
    <lifetime>18:00 7/15/16</lifetime>
    <location>36.886, -76.304021</location>
    <range>5mile</range>
    <distribution>uniform</distribution>
    <count>150</count>
    <resource>
      <name>Accelerometer</name>
      <complimentary>
          <freq>20</freq>
          <direction>gravity</direction>
      </complimentary>
    </resource>
    <resource>
      <name>GPS</name>
    </resource>
    <resource>
     <name>Camera</name>
     <priority>optional</priority>
     <count>50</count>
     <complimentary>
         <resolution>10M</resolution>
         <framerate>30</framerate>
         <flash>1</flash>
         <c_1>facedetection</c_1>
      </complimentary>
    </resource>
    <resource>
     <name>Audio</name>
     <complimentary>
       <channel>mono signal</channel>
       <sample_rate>44100Hz</ sample_rate >
     </complimentary>
   </resource>
   <aggregation>
     <r>GPS</r>
     <r>Accelerometer</r>
    <count>40</count>
     <distribution>NW</distribution>
   </aggregation>
   <aggregation>
     <r>Audio</r>
     <r>Accelerometer</r>
    <count>50</count>
     <distribution>uniform</distribution>
   </aggregation>
 <memo> capture faces  as possible</memo>
</dscript >
</xml>
```

FIG. 8: DiScript Sample

Request Language

Discovery Script (*DiScript*) pronounced as "dee script", an XML-based language to express the need for a set of resources, and all pf the details around their features[1]. It has three parts: *DiScript Initialization*, *Resource Specifications*, and *Resource Aggregation* as shown in Fig 8.

***DiScript* Initialization** is the set of attributes that define the request (i.e. user, lifetime, operation range, count, and resource distribution). *User*, holds *Kinaara*'s ID for the user in creating the script. *Lifetime*, holds the expiry date of the discovery process. The start date is not included as a parameter, since we assume that sending the script is the kickoff date. Although, the script can be composed any time, but once it is dispatched, it will be in action. The dynamic nature of the edge networks would impose the needles overhead keeping a script on mobile nodes if dispatched before its designated execution time. *Location* is the GPS location of the origin where discovery should be performed. Then, *range* and *distribution* to declare an area of operation in miles starting from the origin, and the distribution of devices to discover in this region, respectively. For example, we can request to locate samples from the North West area from origin as shown in Fig 8 line 4. *Count*, mentions the required number of each upcoming resource unless overridden in the resource tag. Finally, *memo* (line 49), is a message sent to device owners, in case a verbal message is required.

Resource Specification, is the core part delivered by *DiScripts*, which details the required resources with their features. Each script should hold at least one resource, otherwise it is discarded. Each resource has two types of features: *Basic*, which includes common features by any resource type (i.e. name, count, priority, and distribution); and *Complimentary*, which are resource-specific features (e.g. only camera has frame rate feature). *Basic features*, are included in every resource. It consists of *name*, *count* of the units to look for, and *priority*, which states if the resource in question is mandatory to find or is optional, and the *distribution*, if a specific sensor has to override the selected option in initialization. *Complementary features* are resource-specific features which differ from one resource to another. For example: Camera (Video, Pictures), has encoding bit rate, orientation, size, frame rate, focus area, focus mode, face detection, snapshot, flash mode, light exposure; Sensors have type (e.g. accelerometer), frequency, accuracy, range, delay, direction

[1]In the future, we will provide a user friendly GUI to generate *DiScripts*

(3-axis sensors) or all three axes; Audio has bit rate, channels, and sampling rate; Network Status, which has interface to monitor (e.g. Wi-Fi, Cellular, or Bluetooth), and bandwidth; and GPS reporting frequency, and provider [134].

The **Resource Aggregation** tag is used to inform the mediator about what resources are required to be collocated one a single device. Later in this section, we will explain how this will reflect on the actual device lookup process.

Fig 8 shows a sample *DiScript* used to recruit 150 resources of four types to monitor an earthquake in action. It starts by initializing the *DiScript* (lines 3-8, and 49). Then, it request four resources: Accelerometer (lines 9-15), GPS (lines 16-18), Camera (lines 19-29), and Audio (lines 30-36) respectively. Within each of theses, a different complementary configuration applies. For example, the camera is required to provide a minimum of 10Mb resolution, 30fps, have its flash working, and face detection capabilities. Finally, two aggregation features (lines 37-42 and 43-48) specify subsets of the request which are needed on one device.

TABLE 3: Search Key Prefix

Feature	**Value**
Count	4 bits, 4 devices per increment
Priority	2 bits dividing task to four levels
Distribution	4 bits each representing a direction (N, S, E, W), with 0000 as uniform
Device	1 bit, "1" means resources on a 1 device

4.5.2 RESOURCE LOOKUP

Fig. 9 depicts *Kinaara*'s resource lookup algorithm. The *mediators* initiate lookup comparing chunks from the search query against its similarity entries. Then, they hop to the most resource similar device (line 3). On each hop, *Kinaara* checks the resource availability and searches the similarity entries for similar devices to visit (lines 4-7). Available resources are held for a *Holding Period*, waiting for an application to execute; otherwise they are free for other resource requests.

In the case of finding all or a subset of the request on a device, the following operations are preformed: *first*, the resources found by the request are removed by eliminating their portion of the search query; and *second*, the selected resources are temporarily marked as *on-hold* until the mediator decides on whether to recruit or not. During the holding time, these resources are viewed as busy by other resource

```
 1: function lookup(Node n, SearchKey s, Path p)
 2:   loop r = s.resources [1 to m]
 3:     if (n.checkAvailability(r))
 4:       s.remove(r)
 5:       n.remove(r)
 6:       p.add(n,r)
 7:       r.hold(t)
 8:     end if
 9:   end loop
10:   if (not empty (s.resouces))
11:     lookup (n.successor, s, p)
12:   else
13:     return p
14:   end if
15: end lookup
```

FIG. 9: *Kinaara* Lookup Algorithm

requests. If these resources are not utilized by the mediator, they become available after a *holding period* timeout. *Finally*, this procedure is repeated with the upcoming similarities until forming a path of devices is formed to fulfill the resource request (lines 2-9). The lookup algorithm has two outputs: *Path Length*, the number of devices visited to match the resource request; and *Discovered Devices*, whose resources are *on-hold* waiting for recruitment.

In designing the lookup operation, we had to consider two aspects, the similarity function and lookup style. *Similarity function* compares search strings with device keys as shown in the lookup algorithm Fig. 9, where we compared multiple token comparison algorithms and chose the Hamming distance for consistency and low latency. *Lookup style* defines how to select the next hop between proximal cluster devices. It can be either looking through the current node's similarity entries one by one or the closest entry in each visited node until the path is established. The former

is a Breadth First Search (BFS), while the latter is Depth First Search (DFS). As the similarity between devices and the number of hops are inversely proportional, *Kinaara* favors the BFS approach, since it utilizes the spatial locality created on the *similarity table*.

4.5.3 MEDIATOR ALLOCATION

In case an application needs to borrow resources from other mediators, the current mediator searches its mediator similarity table for the corresponding resources and selects a target. The target mediator receives a request similar cloud server's resource request, except that the source is another mediator. Mediators comply with such requests, as explained earlier, and report their resources back to the requesting mediator. The requesting mediator appends the returned results to the list of *Discovered Devices* to be used by the edge application.

4.6 IMPLEMENTATION AND EVALUATION

In this section, we evaluate *Kinaara* through simulation, using two kinds of experiments:first, we evaluate *Kinaara* against similar approaches and validate its design choices (e.g. similarity function, lookup techniques); second, we focus on *Kinaara*'s performance under various conditions including varying the percentage of available network resources, changing the number of service requests received per minute, impact of real mobility scenarios, testing *Kinaara* under intensive mobility patterns, and monitoring the impact of holding a resource on the following service requests.

In our experiments we used the following set of parameters: *Network Size*, the total number of devices in the network; *Path Length*, the total number of devices visited till fulfilling a service request; *Discovered devices*, a subset of *Path Length* sufficient to fulfill the service request; *Request Success*, the number of discovered resources as a percentage of the requested ones; *Request Rate(RR)*, the number of received service requests per minute; *Holding Time*, the period used to label a resource unavailable to other service requests; and *Mobility Rate*, the percentage of available resource joining or leaving the experiment.

Throughout our experiments, we simulated devices carrying four resources each, given that we support 16 types of resources. Also, our simulated service requests ask for 2% of the network resources per request. Note that all results presented are the average of 100 operations, each simulating one day (1440minutes) in length. Our key

findings include:

- *Kinaara* is as efficient as Chord in discovery with 70% shorter path length.
- *Kinaara*'s distributed design results in ≈50% smaller path length, compared to the *Centralized* scheme.
- The impact of resource scarceness decreases as the network size increases.
- An increased *Request Rate* (RR) per minute can significantly affect the *Request Success*.
- Mobility affects discovering the first resource holding device, and then the similarity table smoothly finds others.

4.6.1 SIMULATION PLATFORM

Kinaara Simulator

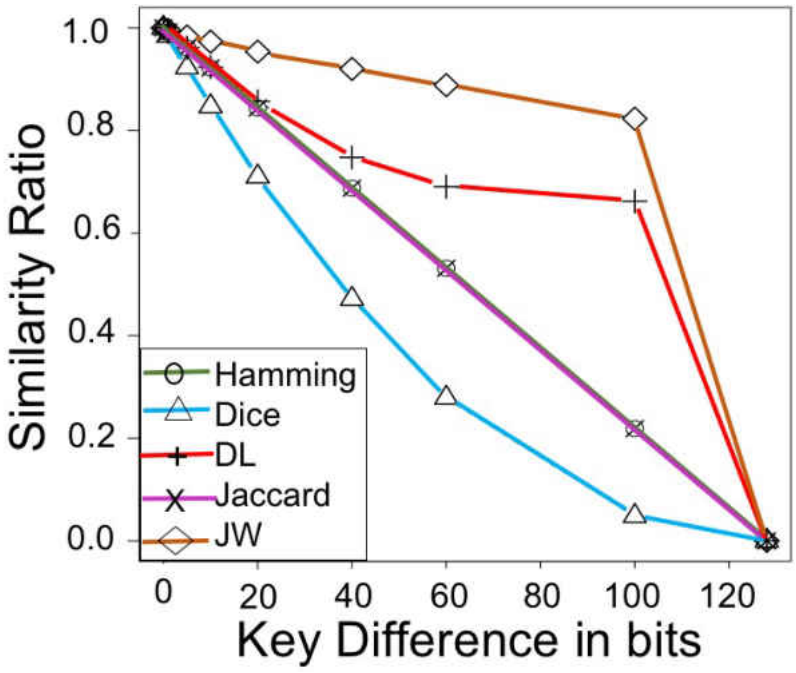
The *Kinaara* simulator is a Java-based simulator consisting of ≈2500 lines of code with no external dependencies to carry all the features explained earlier in the text. The implementation was designed in a modular way (27 classes) for future extensions. We executed our simulator on two machines: First, i7 2.6GHz quad core, with 4GB RAM running Ubuntu 14.04; second, an i7 2.2GHz quad core, with 8GB RAM running OSX 10.9.

Dataset

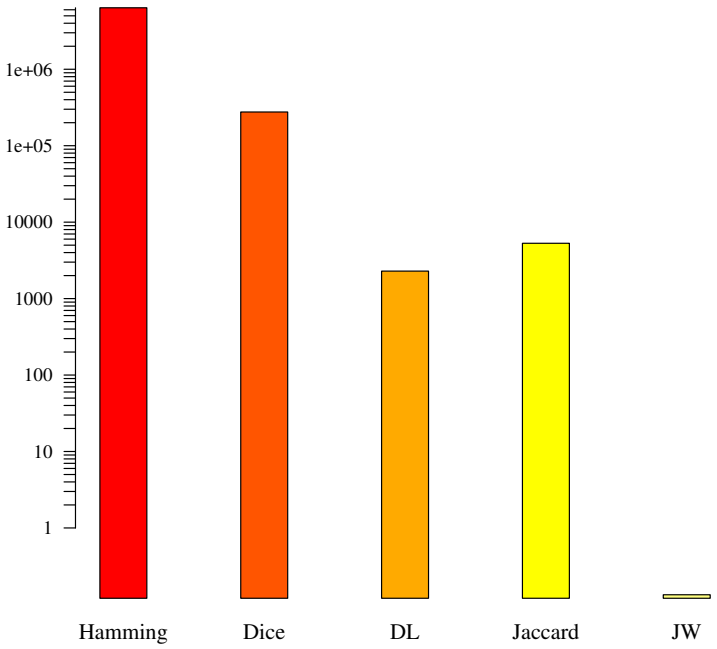
We used WiFiDog, an extensive 6-year dataset of user mobility traces from public Wi-Fi Access Points (APs) [135]. It contains 2,177,835 records of data with 149,861 users moving between 345 Wi-Fi APs. An average of 5732 users visited each AP, each spending an average of 78min/AP. Also, on the average, each user visited 14 different APs. In our simulations, we assume the WiFiDog users are *Kinaara* edge devices and the mediators are deployed at the Wi-Fi APs.

Reference Approaches

IWe compared *Kinaara* to two alternative approaches *Centralized* and Chord [93]. In *Centralized*, *mediators* maintain a dataset of all of the edge devices in a cluster,



(a) Similarity Ratio



(b) Throughput (ops/sec)

FIG. 10: Comparing 5 Similarity Functions to identify the one suitable for *Kinaara*

on a single table, without using the ring structure. *Mediator* can lookup his dataset to locate devices, but must query them sequentially for availability. Assuming that *mediators* up-to-date knowledge of resource availability would fit the purpose of discovery, but would be impractical in predicting the application's execution time. This is required to determine the resource availability for hosting edge applications. *Chord*, a P2P approach to locating files over a large number of nodes, and allow a node to look up a query through hopping to one of its known nodes, with no knowledge of inter-node similarity. Since Chord has a similar strategy to *Kinaara*, we compared them to evaluate the similarity table and the resource based keying design.

4.6.2 *Kinaara* CLUSTERS IN WI-FIDOG

To understand the need for *Kinaara*, we analyzed the Wi-FiDog dataset, for real-time device coexistence. In *Kinaara*, resources are related to the number of available devices; hence, we needed to explore the device availability. Wi-FiDog users spend an average of 78min/AP; hence, we split the dataset to time intervals nearly half this time, i.e., 40mins, and studied those carefully. Our findings showed ≈12K time intervals, *Kinaara*'s ring holds at least 1000 distinct devices (8000 resources) from collocated APs, either through single or multiple mediator scenarios. This means that for those time intervals, the *Kinaara* ring would experience join and leave operations but at least 1000 devices stayed for the whole period. Investigating the APs in the corresponding intervals, we collected those resources from as low as 9APs, which is a low number in enterprise networks, e.g., a mall. Results from such real traces as Wi-FiDog shows the feasibility and the need for a distributed approach to managing those resources.

4.6.3 SIMILARITY FUNCTION

We examined five token based similarity functions: Hamming Distance [136], Dice's Coefficient [137], Damerau-Levenshtein (DL) [138], Jaccard [138], and Jaro-Winkler [138].

In this experiment, we started with two identical 128-bit strings. We had 128 iterations to alter one of them, each flipping one random bit that had not been previously flipped. At the end of each iteration, we compared the two strings using the functions mentioned above. FIG. 10a depicts the string comparison at each iteration. Although Hamming distance and Jaccard were consistent with the key

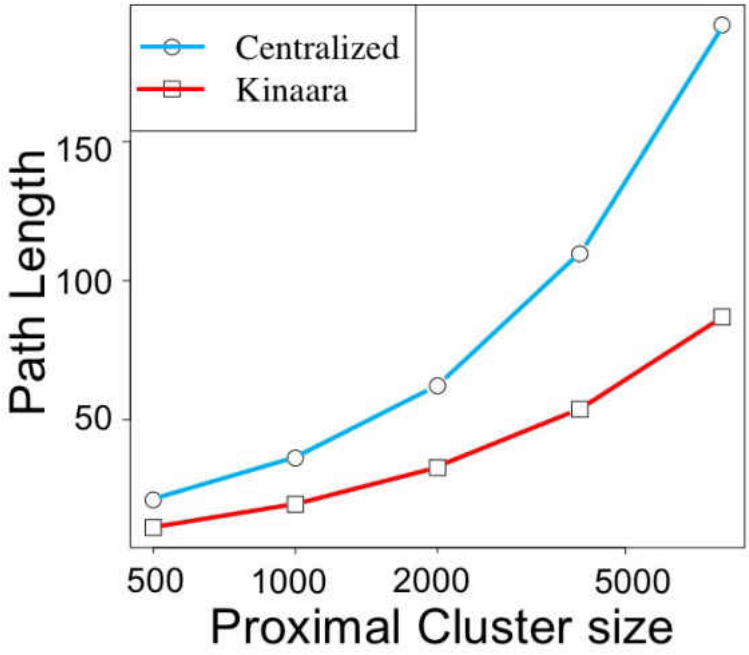


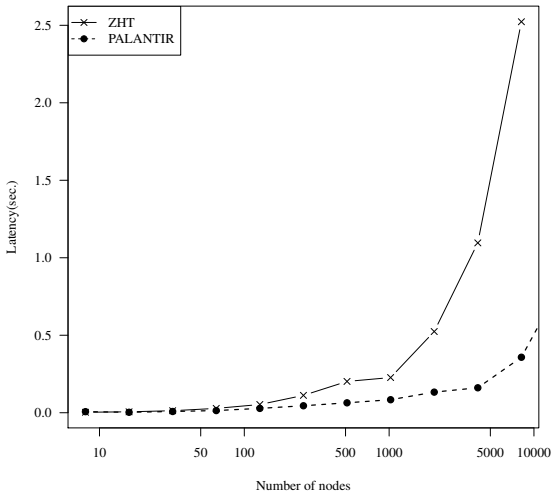
FIG. 11: Centralized Vs. *Kinaara* Resource Discovery

changes, we chose Hamming distance for its throughput superiority as shown in FIG. 10b.

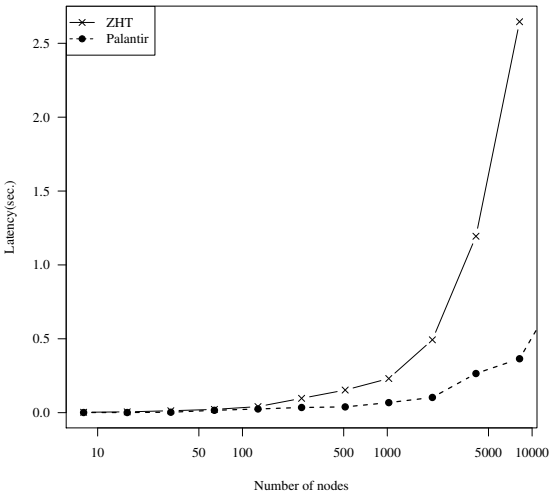
4.6.4 CENTRALIZED VS. DISTRIBUTED RESOURCE DISCOVERY

In this experiment, we compared *Kinaara* to the Centralized approach explained earlier. Centralized places all of the logic on the mediator without using rings or similarity tables. When receiving a resource request, the mediator shortlists devices with the required resources and contacts them one by one to check their resource availability. In this experiment, we mimic real-life scenarios, in which the mediator has no control over the application execution; hence, it has no way to predict if a resource is busy or has finished executing its application. In this experiment, we modeled a busy resource as the summation of two parameters: (a) a *holding time* of 2 minutes and (b) an *app execution time*, a random period between 1-10mins.

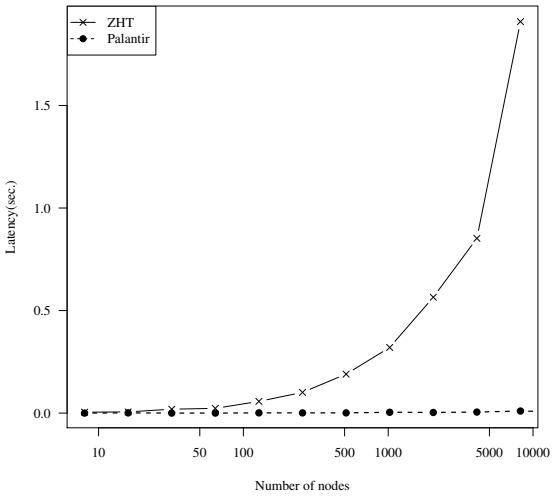
FIG. 11 shows that *Kinaara* visited fewer number of devices, as *Centralized* has higher failure rate from visiting devices with busy resource. FIG. 11 shows that *Kinaara* visited fewer devices, e.g., 40% in 2K resource cluster, with 90% *request success*.



(a) New Devices Joining

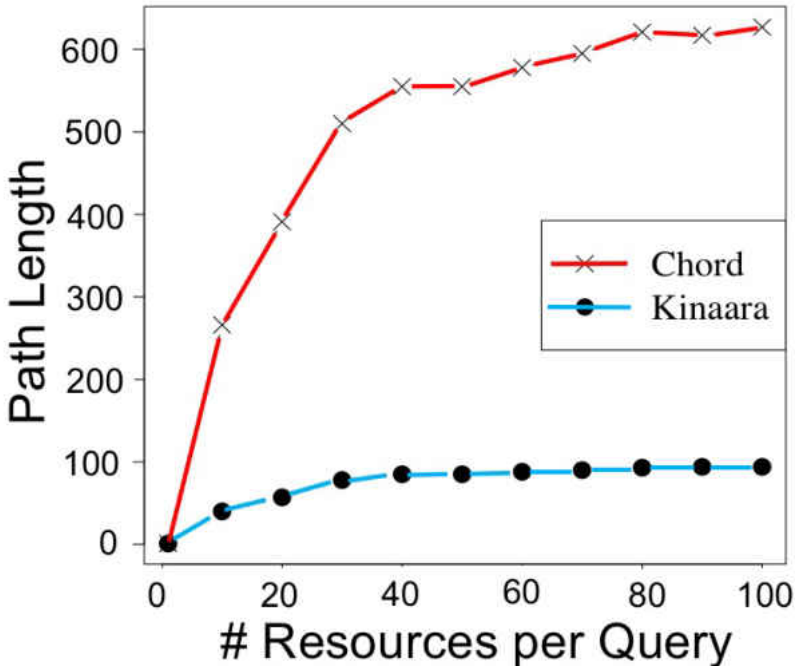


(b) Resource Lookup

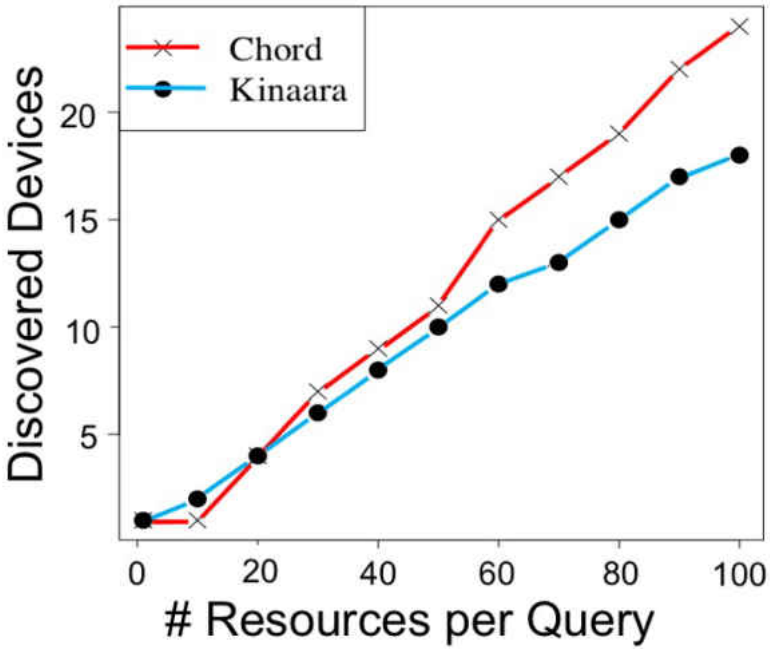


(c) Remove Devices

FIG. 12: Basic Operations: *Kinaara* vs. ZHT [1]



(a) Path Length



(b) Discovered Devices

FIG. 13: *Kinaara*'s selective lookup vs. Chord

4.6.5 BASIC OPERATIONS

In this section, we evaluate three basic operations any discovery approach has to handle: add node, remover node, lookup. Unlike Chord, which has no latency evaluation, *Kinaara* is designed to fulfill low latency requirements. Thus, we compare *Kinaara* to ZHT, which is known for its low latency (i.e. 1.1ms basic operations). However, running ZHT on our machines instead of the IBM Blue Gene 32k-core showed a performance degradation of ≈200X. Also, ZHT generated a large number of threads (proportional to the network size), that prevented our evaluation from exceeding the 8K node network on our machines, and probably degraded the performance.

FIG. 12 shows how our basic operations outperformed ZHT. FIG. 12a and FIG. 12b, show almost the same latency for Join, and for lookup operations, respectively. However, our remove in FIG. 12c shows almost a constant value to delete the node from the ring, leaving its appearance in other similarity tables to be fixed upon discovery.

4.6.6 EFFICIENT PATH LENGTH

In this experiment, we investigate the number of device hops needed to fulfill a search query. We compare *Kinaara* to Chord's open source code [93] with both subject to the same input. *Kinaara* selects its next device to look for resources through resource similarity comparison with the search query, while Chord hops sequentially between devices until it finds its target. Using randomly generated queries, we performed this experiment on a 1K device cluster (8000 resources, a practical number from § 4.6.2).

FIG. 13b shows that Chord and *Kinaara* to discovered almost the same number of devices. *Kinaara*, however, visited fewer devices, which resulted in a customized resource lookup that could reach 70% fewer device visits as shown in FIG. 13a. This experiment shows that the similarity table significantly decreases the number of device hops per search query.

4.6.7 CLUSTER ALLOCATION

In this experiment, we measured the impact of a congested cluster over *Kinaara*. This occurs when resources are either scarce or are pre-allocated to other applications.

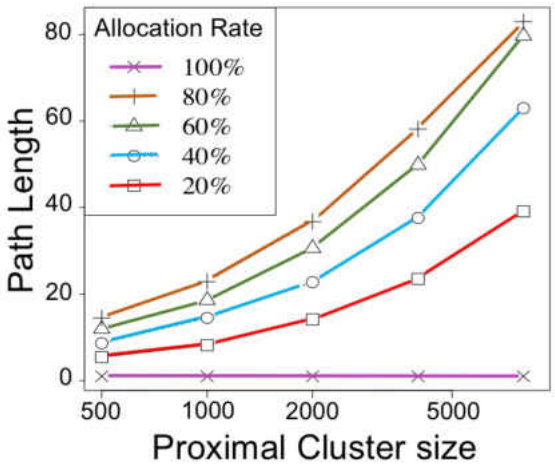


FIG. 14: Cluster Allocation, impact of increasing resource usage ratio on discovery

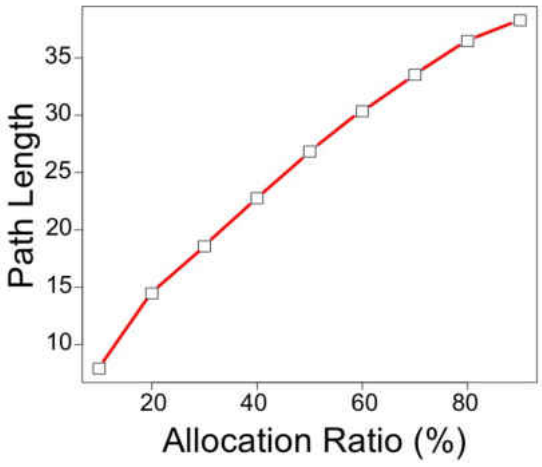


FIG. 15: Path Length under different Cluster Allocations on a 2k resource cluster.

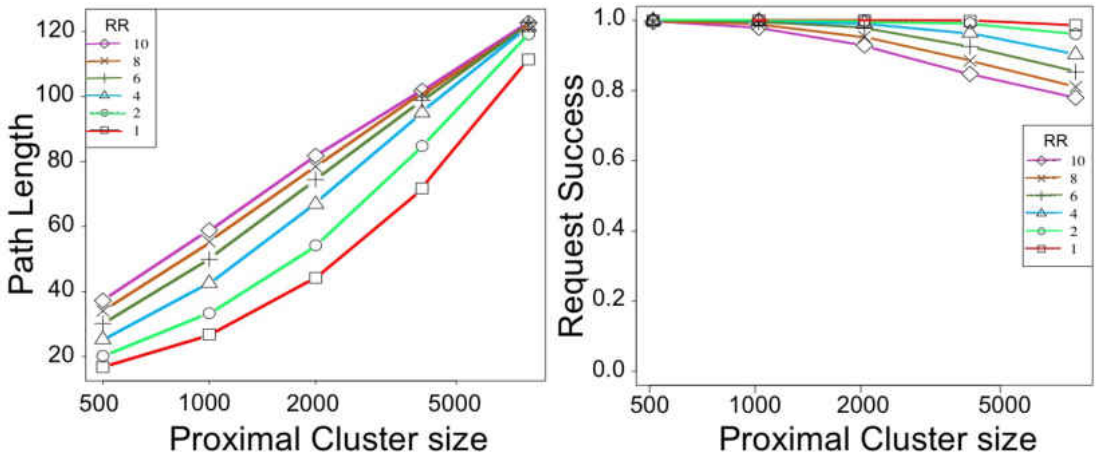
In this experiment, we used variable cluster sizes, each was subject to an assumption that a portion of its resources was pre-allocated.

FIG. 14 showed an increase in the path length to fulfill the same search query as the initial assumption of decreased available resources. We monitored a decreasing rate of the path length increase as shown in FIG. 15, while achieving the same *request success*. This shows another benefit of using a key that conveys available resources and a similarity table that rectifies the scope of lookup to the most similar peer, not to just any peer.

4.6.8 RESOURCE REQUEST RATE (RR)

In this experiment, we monitored the impact of multiple *Request Rates* (RR) on path length and request success.

FIG. 16a shows the path length to increase as the *request rate* increases, however, the spaces between the curves decrease implying a controlled *path length* increase. The *path length* behavior came along with a decrease *request success* due to discovery failure under intensive requests as shown in FIG. 16b. Increasing the *request rate* holds a large number of resources, for example *request rate*=10 on a 2000 resource network resulting in 400 resource request per minute, which leads to requesting the whole network's resources in 20 minutes. Therefore a *mediator* has to control the *Kinaara request rate* while considering the number of resources available in the cluster.



(a) Path Length (b) Request Success

FIG. 16: Impact of Request Rate (RR) on resource discovery

Also, it has to report to the *cloud server* that a *proximal cluster* is receiving many requests asking to target other clusters.

4.6.9 MOBILITY RATE

In this experiment, we measured the impact of variable mobility patterns on *Kinaara*. A mobility pattern involves various devices joining and leaving the cluster that does not belong to the WiFiDog dataset. This means that within each time interval, i.e., 1 minute, a percentage of the cluster devices move in or out. This scenario shows the impact on discovery by *Kinaara*'s device leave strategy (§ 4.4.2) that does not remove entries in similarity tables for devices that left the cluster.

We investigated multiple mobility patterns, i.e., 10-80%, and all had similar impact on *Kinaara*. FIG. 17 depicts the zero and 20% patterns. We witnessed low impact on clusters having up to 3000 resources, while path length increased between 20-45% in larger clusters. Hence, mediators should consider mobility prior to expanding the cluster size from neighboring APs.

4.7 CONCLUSION

In this work, we presented *Kinaara*, a distributed resource discovery solution for mobile edge networks. First, we proposed the concept of keys encoding the resources of each device and designed and implemented the Resource Encoder that

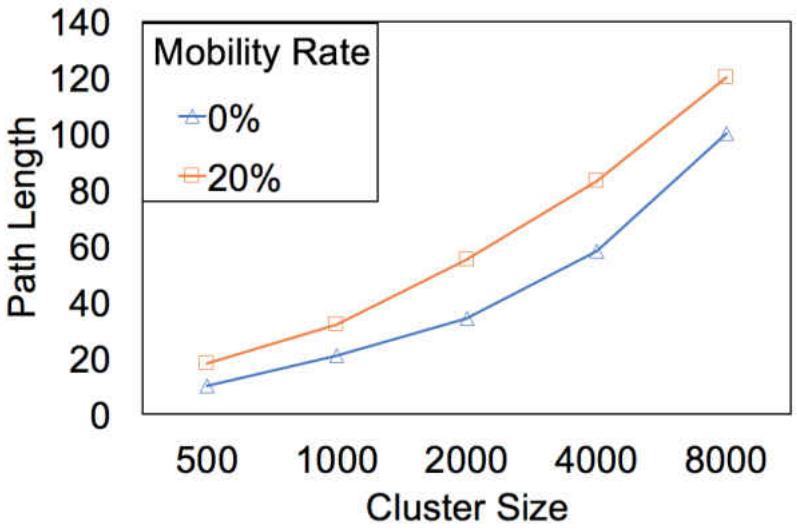


FIG. 17: Mobility Rate and its impact on discovery.

maps resources to keys. Second, we used keys to sort devices on a logical ring structure, each with a data structure, i.e., similarity table, pointing to devices with similar keys, i.e., resources. Third, we leveraged previous modules for an optimized resource lookup. Finally, we evaluated *Kinaara* against other approaches and under stress testing conditions. Our evaluation showed that *Kinaara* efficiently discovered resources per incoming request, showing *Kinaara* to be 70% better than Chord and 40% than Centralized. In terms of mobility, *Kinaara* had low overhead at high mobility rates in clusters up to 3K resource compared to stationary modes.

On the other hand, our results showed some lessons to be learned. First, they showed the need for a smart load balancing system at the cloud to avoid overwhelming a specific mediator with a high request rates, more than the cluster can accommodate, which degrades the request success rate. Second, we learned, from the mobility experiments, is the limit that *Kinaara* could scale to, i.e. 3000 resources per cluster. While creating proximal clusters and sharing resources between mediators, we need to keep the resources within a cluster below the threshold that would lead to a degraded performance.

CHAPTER 5

BOSS: BLUETOOTH OPEN SOURCE STACK TO FACILITATE EDGE COMMUNICATION

The dynamic nature of Mobile Edge devices requires a flexible communication strategy. Bluetooth showed up as a prominent candidate for communication; however, it has some limitations in terms of communication latency, data rate, and interference [28]. All of these limitations are not addressable, so far, due to the absence of an open source firmware, as explained earlier in Chapter 2. In this dissertation, we propose *BOSS*: a Bluetooth Open Source Stack to tailor Bluetooth for edge communication and grant researchers access to low level Bluetooth layers [139]. Unlike other IEEE standards, e.g., Wi-Fi, Bluetooth is always considered a black box to be used as is without the ability to customize its module. This negates the dynamic nature that edge systems should have. We designed *BOSS* to adhere with Bluetooth v4.0 core specifications in order to provide the community with a reliable open source version. In this dissertation, we present the challenges to realizing the unique Bluetooth features, and ways to address them, in addition to a full implementation of Bluetooth core specifications detailed in the IEEE standard [107].

5.1 OVERVIEW

Communication is a major concern in any system; however, there is a wide belief in the research community that Bluetooth v4.0 is the best candidate to host edge communication. Not only because of the low power consumption of Bluetooth v4.0, but because its efficient design fulfills the needs for mobile edge in terms of peer discovery, establishing a connection, and device-to-device data exchange. Moreover, nowadays all wearables are fabricated with Bluetooth support.

Unlike other communication protocols, e.g., Wi-Fi and Zigbee, Bluetooth has no open source firmware. During the last decade, there has been a the huge community contribution in implementing, and testing the Wi-Fi protocol lower layers, some of which made its way to the IEEE standard (e.g., MIMO). On the other hand, researchers tend to use Zigbee for one-to-one applications despite its requirement for

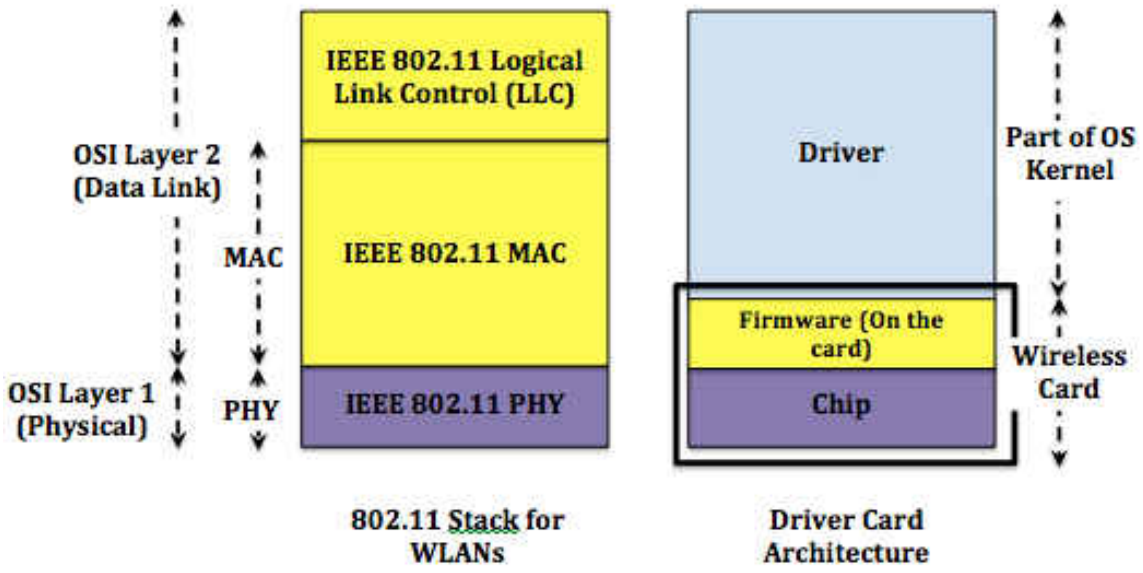


FIG. 18: Communication Stack Openness in IEEE 802.11 (Wi-Fi) vs Bluetooth

an extra hardware, because of its open source structure. FIG. 18 shows the how the communication layers are distributed, both in Wi-Fi and in Bluetooth. Unlike in Wi-Fi, in Bluetooth, all of the firmware design and implementation is enclosed on a chip controlled by the manufactures, who are less likely to either expose their design or address issues beyond their scope. The goal of this work is to decouple the Bluetooth firmware from hardware chip to customize Bluetooth for edge communication for grant researchers' control over these layers.

In this dissertation, we present *BOSS*, a Bluetooth Open Source Stack to grant access to lower layer Bluetooth core specification. Bluetooth's lower layer has different baseband capabilities compared to other communication protocols, e.g., inquiry and frequency hopping. These features are enclosed in the firmware provided by each manufacturer. *BOSS*, following the Bluetooth v4.0 standard, provides an open source implementation of Bluetooth features. For example, the Bluetooth inquiry process starts with a master device sending requests to locate peers over the 79 channels of the 2.4GHz spectrum one by one. Bluetooth sends an inquiry request on the current channels and waits 625μsec for a response from another device. Upon receiving a response both devices synchronize their frequency hopping pattern. If none is received throughout the waiting period the master hops to the next channel until it covers the whole spectrum in one inquiry cycle. The hopping pattern, is assigned by the

frequency hopping algorithm with parameters from both master and slave. Upon completing the device discovery process a master and a slave associated together and starts their data communication. *BOSS* provides an open source implementation to this process and to others.

BOSS is hosted on Ubertooth One hardware: an open source 2.4GHz wireless development platform [140]. Ubertooth is a cheap hardware with assembly instructions that from scratch is available online. Ubertooth is designed to perform passive packet sniffing tasks for spectrum monitoring. However, its hardware is capable of transmitting and channel hopping, in order to support the Bluetooth hardware standard pre-requisites.

BOSS, upon public exposure, can be used by research communities. The rise of edge computing can benefit from *BOSS* in creating a communication strategy customized for each application. Moreover, researchers trying to resolve the interference resulting from the co-existence of Wi-Fi and Bluetooth on the same device will find a practical solution, other than simulation-based approaches.

The focus of *BOSS* is to provide an open source Bluetooth stack for researchers following the Bluetooth standard design details. *BOSS* has the following contributions. It offers:

1. A pioneering attempt to provide an open source repository for Bluetooth core specifications.

2. A detailed design of the Bluetooth stack modules.

3. Hardware-level implementation and evaluation for Bluetooth device discovery and packet transmission.

5.2 BOSS IN ACTION

In this section, we provide sample use cases for *BOSS* that are not in any way inclusive.

Wearable Communication

Wearable communication has a significant presence on the edge. Wearable devices are used to collect data and send it to a host device for processing. If more than one device is used every device performs its communication task to the host, which

collects all of the data and performs its own processing. This includes plenty of control message exchange to establish communication between each single device and the host. Moreover, there is the Bluetooth limitation in communicating with a single device at a time. This not only leads to a queue of devices waiting to communicate with a host, but to a delayed processing until all of the data is collected [141].

Multi-hop Communication

Bluetooth is originally designed for peer-to-peer communication. However, with the release of v4.0 along with IoT evolution, Bluetooth can support multi-hop routing in a multi-node edge network to handle a huge number of devices. The IoT mesh network requires the development of scalable, reliable, and efficient cross layer protocols for Bluetooth networking layers. Open source Bluetooth stack is needed to enable prototype development and to encourage researchers to work on this domain.

5.2.1 CUSTOMIZABLE DEVICE DISCOVERY

Bluetooth device discovery with its frequency hopping mechanism has always been a delaying component in Bluetooth v2.0 communication. The release of Bluetooth v4.0 with BLE features has modified the discovery component to hop over only three channels (i.e., advertising channels), instead of 79. This has decreased the inquiry delay time from 10ms to 3ms. However, this came on the data rate to also decrease to ≈200kbps instead of 1-3Mbps. In addition, the communication range was almost halved to ≈50m [142].

Despite the enhancements of Bluetooth v4.0, many applications finds it not suitable. For example, vehicular applications due to their high mobility nature require an extended communication rate and range. For example, Bluetooth is used to estimate vehicles' speed and road congestion by collecting data from Bluetooth devices on board of vehicles crossing a specific location [143, 144]. Bluetooth v2.0 are likely to spot them for ≈35sec, while v4.0 devices will only spot it for ≈10sec. In this short a time devices needs to collect multiple responses from Bluetooth devices on vehicles. In such a scenario, the need for a mixed features from Bluetooth v2.0 and v4.0 shows the need for an open source Bluetooth stack.

BOSS shows itself to be a perfect candidate to tune Bluetooth to application specific needs. This feature is not currently available due to the absence of an open source Bluetooth stack, which limits the spectrum of edge applications.

5.2.2 BLUETOOTH TESTBED

The wide availability of 802.11 testbeds (e.g. USRP [145] and WARP [146]) have opened research directions and innovations. On the other hand, Bluetooth with its community acceptance still lacks the openness of 802.11. The presence of a testbed will motivate researchers to modify and address deficiencies within the Bluetooth standard.

Hardware co-existence of Wi-Fi and Bluetooth chipsets is in almost all mobile devices. However, it still causes signal interference that degrades both performances. The research community has tried hardly to address this problem, but the lack of open source Bluetooth firmware has directed this research to simulation-based studies [147, 148]. As a result, only Bluetooth devices manufacturers can work on this, eliminating the efforts of a wide majority of networking researchers.

Frequency hopping is considered a point of delay in Bluetooth communication. Attempts have been made to enhance the process through simulation, but none of this has made it through to real devices due to a lack of real-life evaluation. Manufacturers can hardly risk their products for a less than perfect solution.

5.2.3 BLUETOOTH ADAPTER

Attempts to open Bluetooth to the public started using the tools GNU-Radio [149], and USRP boards [145]. Despite the technical challenges, user-friendly tools have been developed to easily attempt the technical challenges. However, the main burden for community use have been the hardware costs, which can reach thousands of US dollars, and the large size. Hence, the Ubertooth team began working and released Ubertooth Zero, followed by Ubertooth One.

Ubertooth One

Ubertooth One is an open source 2.4GHz wireless development platform [140]. Ubertooth One hardware shown in FIG. 19 comes with an RF Connector for the antenna, an ARM microcontroller, and a wireless transceiver. Moreover, it features an inexpensive hardware that can be easily or acquired or assembled using the publicly available instructions.

Ubertooth One's hardware is designed to support Bluetooth's transmit/receive

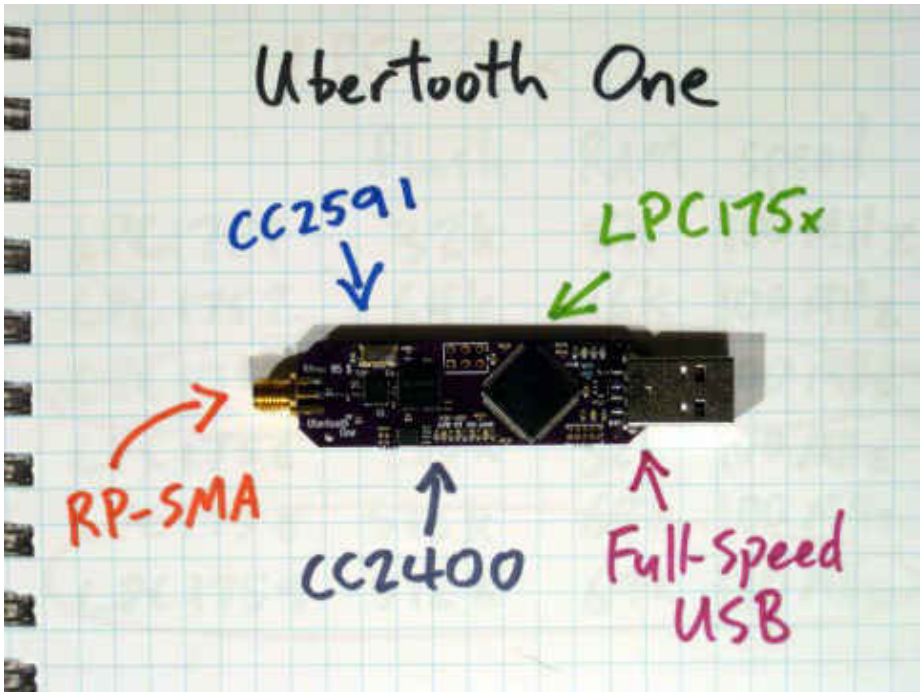


FIG. 19: Ubertooh One Hardware Design

features. The main objective of the Ubertooth project is to develop simple Bluetooth packet sniffing via a USB port with a host application on a PC. In late 2014, Ubertooth One introduced the capability of sniffing BLE packets. Moreover, the Ubertooth team maintains an open source repository for all codes from firmware to applications. Although the hardware is capable of transmission, Ubertooth One does not support it, by any means.

Ubertooth One Architecture

FIG. 19 depicts the 5 hardware modules provided by Ubertooth. I/O is done through the USB, and RP-SMA RF antenna connector. CC2591 is an RF from end to extend communication range. LPC175x is an ARM Cortex-M3 microcontroller to support multiple high-bandwidth data streams. A CC2400 wireless transceiver is used to provide a low power communication over the 2.4GHz band.

In this dissertation, *LAMEN* modifies the firmware to support various Bluetooth features other than packet receiving which is currently supported. The modified firmware is loaded to the LPC175x microcontroller for further usage.

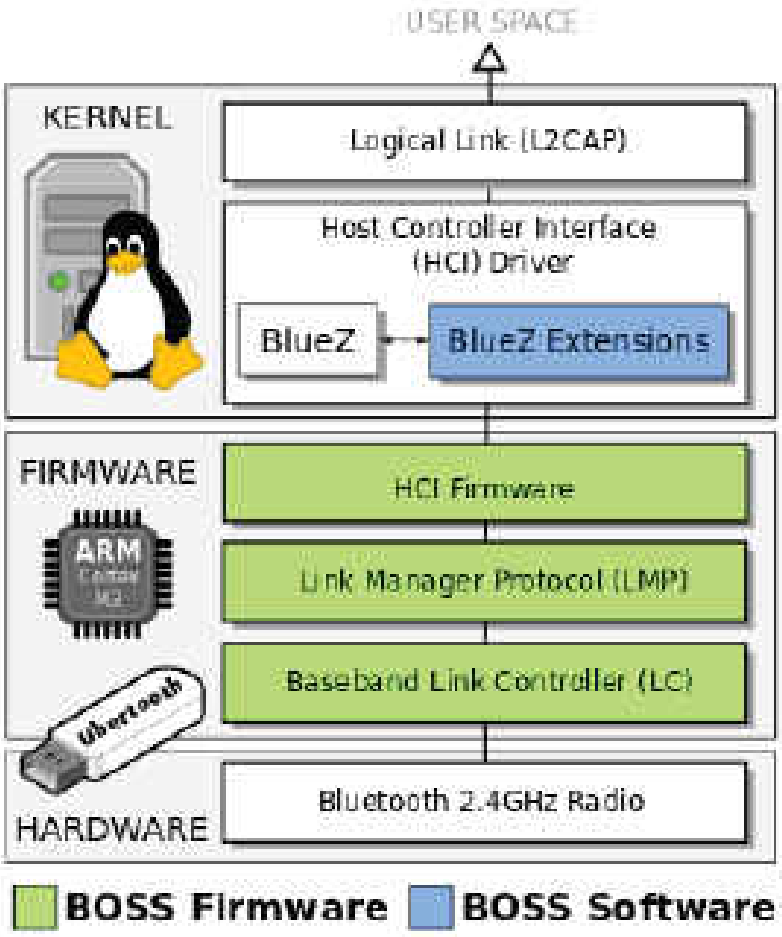


FIG. 20: BOSS Components Architecture

5.3 BOSS PLATFORM

FIG. 20 depicts the modules that *BOSS* is providing towards an open source Bluetooth stack. The *BOSS* contribution comes two parts: the *firmware* shown in green, which is currently locked by manufacturers, and the *software extension* shown in blue, which provides APIs to facilitate the modified firmware usage by the kernel space software.

5.3.1 FIRMWARE

BOSS uses Ubertooth One to introduce firmware modules, as shown in FIG. 21: Baseband, Link Manager, and HCI. In this section, we will explain the contribution of *BOSS* on the corresponding firmware modules.

Fundamental Features

BOSS relies on Ubertooth One to host the open source framework, despite its exclusive support for passive packet sniffing. Therefore, we needed to implement the following fundamental features in order to comply with *BOSS*'s needs.

Transmission: Ubertooth firmware support only packet sniffing but not transmission. Hence, we introduced the transmission feature module on Ubertooth One hardware. Using the Ubertooth transceiver's (i.e., CC2400) data sheet, we could transmit and receive packets on two Ubertooth devices each connected to a different PC. We used an Ellisys sniffer [150] to capture and trace transmitted packets to detect problems (such as CRC calculation problem) and to validate our developed transmission module.

Circuit Switching: Establishing a Bluetooth connection requires fast switching between the transmission and the reception modes. This was not supported by Ubertooth One firmware with no available documentation. *BOSS* provides support and implementation to fast switching functionality in runtime (microsecond level). *BOSS* alternates between the Tx/Rx modes to exchange messages between multiple Ubertooth devices.

Frequency Hopping: Building on both the transmission and circuit switching, we had to introduce both features with continuous change in the channel. This channel modification required handling a large number of buffers in the Bluetooth as it is usually accompanied by a change of mode, either transmission or reception.

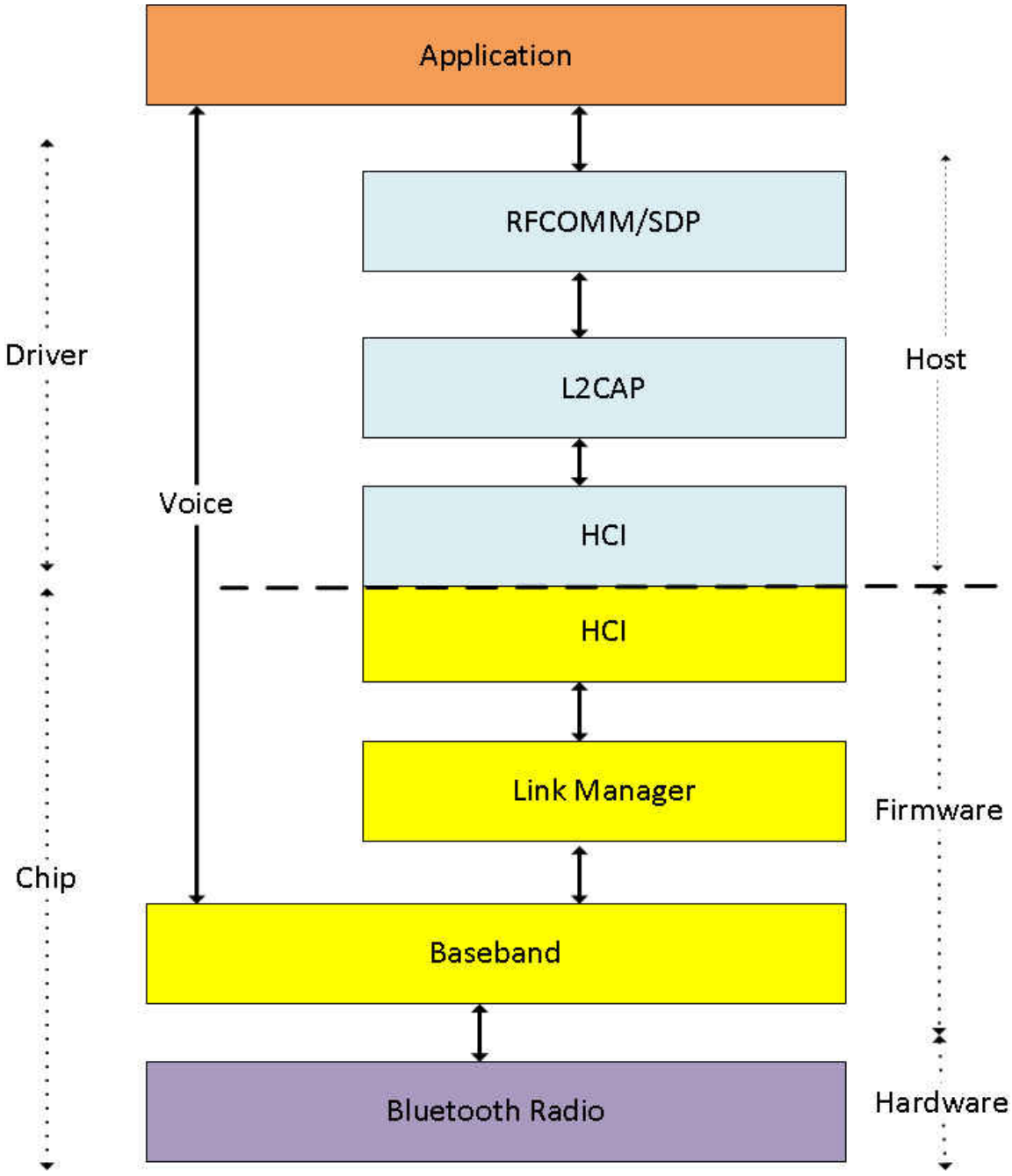


FIG. 21: Bluetooth Protocol Stack

Baseband Controller

The Bluetooth standard requires any host to have a Baseband (BB) controller that supports one of the following functionalities: Basic Rate/Enhanced Data Rate (BR/EDR) only, Low Energy (LE) only, or BR/EDR/LE. In *BOSS* implementation, our contribution was in the Low Energy Mode, as follows:

Bluetooth Clock supports different clock modes (e.g. native clock and master clock), and the corresponding clock management. The Clock is a 28-bit counter with the least significant bit will tick in units of 312.4μsec providing a clock rate of 3.2kHz. Once a Slave receives the master's clock, it adapts its clocks accordingly. Ubertooth provided the basic clocking functionalities and *BOSS* added to it variations of the clocking sequences used to adjusting the transmission and receiving time slots.

Bluetooth Addressing a 48-bit address to uniquely identify the device. The address consists of 24-bit Lower Address Portion, 16-bit Non-significant Address Portion (NAP), and 8-bit Upper Address Portion (UAP). This code is used to generate Device Access Code (DAC), Channel Access Code (CAC), and Inquiry Access Code (IAC). These codes are used in various procedures (e.g. Inquiry process). *BOSS* utilizes the 48bit Bluetooth Address provided by Ubertooth One, unused so far, to generate a unique ID per device in each connection. This enables two or more devices to participate in the ongoing communication without presetting their IDs.

Physical Channels, this feature is responsible for the frequency hopping process, specifying a time slot for transmission, and Bluetooth supported access codes and packet header encoding. This module should carry six modes of frequency hopping supported by Bluetooth: page hopping, page response hopping, inquiry hopping, inquiry response hopping, basic channel hopping, and adapted channel hopping. The hoping mode uses the UAP/LAP, and clock as parameters to decide the next channel in the hopping sequence. *BOSS* implemented the corresponding 2 modes required by the sender and the receiver (i.e. inquiry hopping sequence, and inquiry response sequence).

Physical Links are established between master/slave or piconet members to be used in communication.*BOSS* enables the master to maintain a list slaves, up to 7, and use their Bluetooth addresses in sending directed messages.

Logical Links provides communication APIs between two of the firmware components. i.e., Baseband and Link Manager. *BOSS* provides the APIs to exchange control and data message between both layers.

Bluetooth Packets is the data structure that defines the information exchanged between Bluetooth devices. It supports the following parts: access code, header, and payload. The Access code is used to limit the number of responses when looking for devices in range. The header provides the type of exchanges message as the Bluetooth supports a large set of message types. Finally, the payload is the data to be transmitted. Ubertooth One supports that packet structure. However, it has a static value for access code in the firmware is not accessible by the kernel space. This constraint has to be relaxed as it limits the communication to devices carrying the same static address.

out of the thousands of packets defined by the standard, *BOSS* support 32 packets, some of which covers the four of the the BLE states:

1. **Advertising state**:
   ADV_IND, ADV_DIRECT_IND, ADV_NONCONN_IND, and ADV_SCAN_IND.

2. **Scanning state**:
   SCAN_REQ and SCAN_RSP.

3. **Initiating state**:
   CONNECT_REQ.

4. **Connection state**:
   LL Data PDU, LL Control PDU, LL_CONNECTION_UPDATE_PDU, LL_CHANNEL_MAP_REQ, LL_TERMINATE_IND, LL_ENC_REQ, LL_ENQ_RSP, LL_START_ENQ_REQ, LL_START_ENQ_RSP, LL_UNKOWN_RSP, LL_FEATURE_REQ, LL_FEATURE_RSP, LL_PAUSE_ENC_REQ, LL_PAUSE_ENC_RSP, LL_VERSION_IND, and LL_REJECT_IND.

Bitstream Processing is responsible for extracting packet information and error correction. In addition to packet whitening, a Bluetooth process performed on the packet's payload and CRC to hide the correlation between the data stream. The whitening process is executed at the transmitting side and reversed on the receiving side. *BOSS* provides packet processing and the whitening process for all the 23 packet types supported.

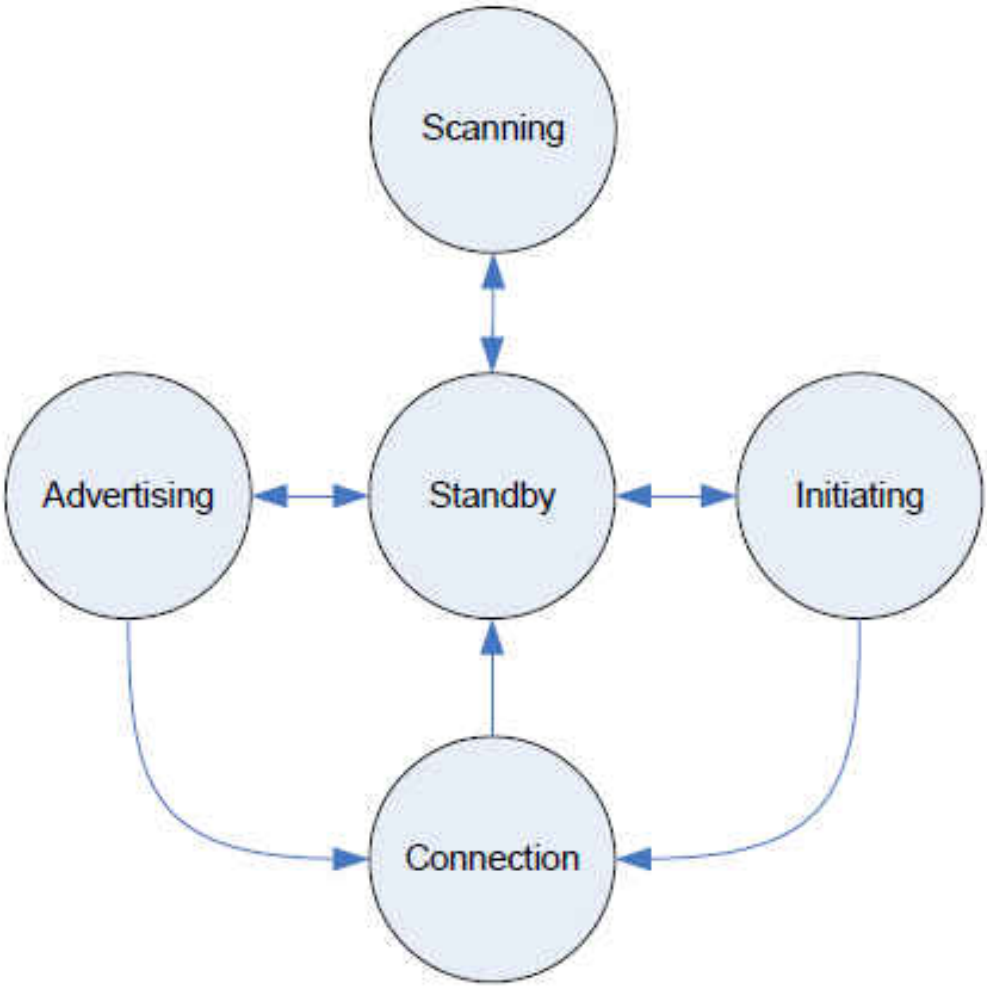


FIG. 22: Bluetooth State Diagram controlled through the link manager in the firmware

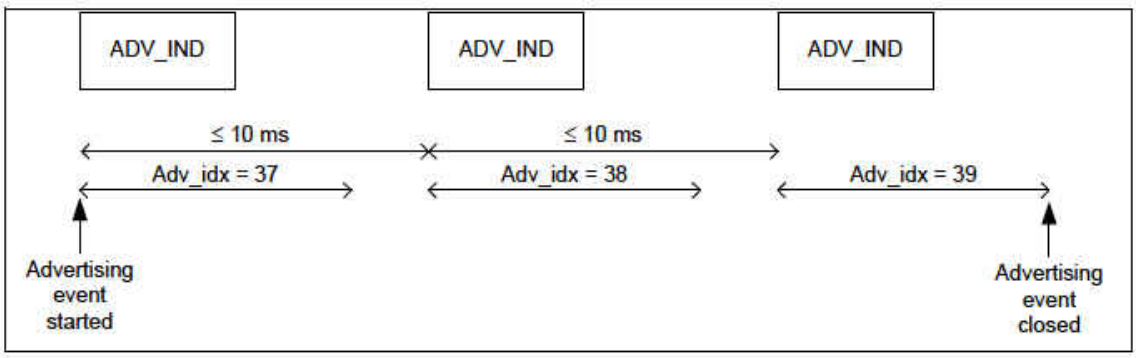


FIG. 23: Bluetooth Advertising State

Link Manager

Link manager utilizes the low level features provided by the baseband to execute the Bluetooth logic features. All components of the link manager were not part of the Ubertooth project and an exclusive to *BOSS*. Link manager consists of three components as follows:

Link Control (LC) manages the transition between the 5 states of Bluetooth connection, i.e., Scanning, Standby, Initiating, Advertising, and Connection. It also provides the interface for link manager functionalities through HCI. *BOSS* supports the 5 states of Bluetooth and the transition between them based on the status of communicating parties, as shown in FIG. 22.

BOSS implemented the following features in every state. The *standby* state is whenever a Bluetooth device is idle, while the *scanning* is sniffing packets but a special frequency hopping mechanism described in the standard. The remaining three states are the ones responsible for communication. A device in the *advertising state*, called advertiser, keeps sending ADV_IND packet to introduce him self to others. Frequency hopping in this state is limited to three channels (37, 38, and 39) with no specific order, every manufacturer can have his own algorithm here, the only specification is that the listing period per channel should be a factor of 612μsec with a maximum value of 10ms. This state, eventually, results in a slave device should the data communication be established.

Initiating state, is the contrary of the *Advertising*, a device performing it is called the initiator. An initiator picks a channel at random and listens for a random period with the same timing specifications as the *advertiser's* time slot. Unless

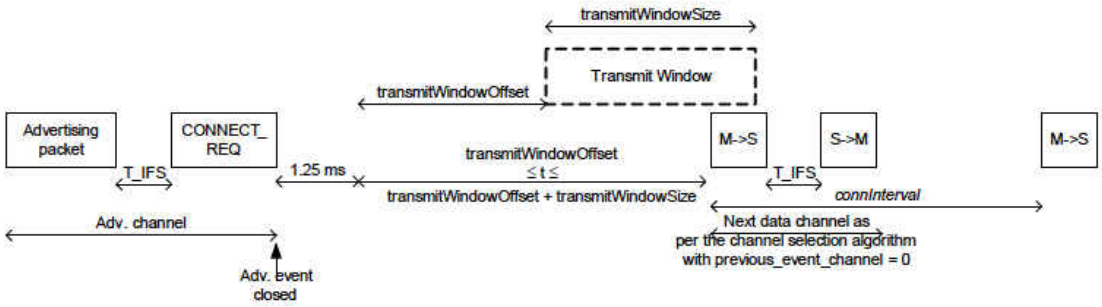


FIG. 24: Bluetooth Connection State

an ADV_IND packet is received the *initiator* hops to the next channel. Once an ADV_IND packet is received the *initiator* replies back with a packet called CONNECT_REQ that includes all the information needs for both devices to synchronize their hops. Also, the *initiating state* yields a master device. At this point both devices enter the final state, *connection* and exchange data packets with by hopping to the same channels until they decide to end the connection. In case any connection parameter has to change, e.g., hopping sequence, both devices use the *flow control* component of the link manger explained later in this section.

Link Manager controls the radio link between two devices through the link manager protocol (LMP). It is responsible for link establishment, collecting device capabilities, and power control. LMP interprets the received signals without propagating them to upper layers. *BOSS* provides LMP messages transfer through the payload. These messages are distinguished by flags in the header.

Flow Control avoids losing packets due to an overflow at the receiver's buffer, bad channels, or frequency hopping errors. *BOSS* uses packet formats in the Bluetooth, e.g., LL_CONNECTION_UPDATE_REQ, to rectify the connection based on the channel status.

HCI firmware

This module is responsible for the communication between the firmware and the host, e.g., BlueZ [108]. Commands sent from the host are filtered and redirected to the corresponding link manager module. Also, the host can acquire hardware status information through this module, e.g., the connection state. *BOSS* provides a set of APIs to communicate with the firmware in a manner similar to that of any Bluetooth

commodity device. In addition to this, it offers the possibility of adding any set of APIs and their corresponding firmware modification.

5.4 TESTING

In order to validate the performance of *BOSS* we carried out one experiment with two Ubertooth devices and two experiments with commodity hardware, i.e., Fitbit [151] and iPhone. Using Ubertooth devices, we were able to exchange control information with each device randomly executing a state from the initiating and advertising. Both devices successfully negotiated the connection and exchanged empty data packets until we ended the connection.

Testing *BOSS* with commodity devices Fitbit and iPhone, the former passed the control and initiated the data exchange with an empty data packet while the latter only passed the control phase. FIG. 25 shows the message exchange between *BOSS* running on an Ubertooth One device and Fitbit Charge 2. *BOSS* entered the session as an initiator by receiving an ADV_IND packet and responding with a CONNECT_REQ. At that point, the session entered the Bluetooth connection state. The two devices exchanged an empty data packet on a data channel (13 in the experiment shown by FIG. 25). The session could not carry on any further, since the Fitbit, at this point, was expecting an encryption negotiation with the other device to carry on with the connection. We envision that *BOSS* will carry a full connection, upon implementing Bluetooth encryption features. FIG. 26 depicts the message exchange between *BOSS* and an iPhone 7. It showed almost the same steps; however, the iPhone connection did not go through any empty data packet exchange, unlike Fitbit, as the iPhone expects encryption negotiation to start with the devices entering the connection state. The difference between Fitbit and iPhone behavior was detected through close monitoring of both device interaction together and with *BOSS* using the ElliSys spectrum monitor [150].

5.5 CONCLUSION

In this chapter, we presented *BOSS*: a Bluetooth Open Source Stack. *BOSS* enables the customization of Bluetooth v4.0 for the specific needs of edge computing. In addition to this, it enables researchers to customize and explore Bluetooth features to fit their applications. Researchers have used Zigbee despite its low market presence compared to Bluetooth, for its code openness. *BOSS* is available to the

FIG. 25: Validating the performance of *BOSS* through pairing and data exchange with a FitBit Charge 2

```
ahesham:~ ahesham$ ubertooth-btle -M58:3a:c3:33:3d:86 -m6b:8e:d3:d6:38:55
InitA set to (master Adr): 58:3a:c3:33:3d:86
H sending Control: 6b:8e:d3:d6:38:55:
AdvA set to (slave Adr): 6b:8e:d3:d6:38:55
  Initiating state - BOSS
r ADVtype(as):40
,  ADVtype(as):00
,systime=1514386674 freq=2402 addr=8e89bed6 delta_t=283.895 ms rssi=-20
Raw pkt:d6 be 89 8e 40 14 86 3d 33 c3 3a 58 02 01 1a 0a ff 4c 00 10 05 07 10 5d b1 9c b5 41 ea
Payload:      40 14 86 3d 33 c3 3a 58 02 01 1a 0a ff 4c 00 10 05 07 10 5d b1 9c b5 41 ea
Advertising / AA 8e89bed6 (valid)/ 20 bytes
    Channel Index: 37
    Type:  ADV_IND
    AdvA:  58:3a:c3:33:3d:86 (random)
    AdvData: 02 01 1a 0a ff 4c 00 10 05 07 10 5d b1 9c
      Type 01 (Flags)
          00011010
              LE General Discoverable Mode
              Simultaneous LE and BR/EDR to Same Device Capable (Controller)
              Simultaneous LE and BR/EDR to Same Device Capable (Host)
      Type ff (Manufacturer Specific Data)
          Company: Apple, Inc.
          Data: 10 05 07 10 5d b1 9c
    Data:  86 3d 33 c3 3a 58 02 01 1a 0a ff 4c 00 10 05 07 10 5d b1 9c
    CRC:   b5 41 ea
Connection mode - BOSS
Master connection is crea
```

FIG. 26: Validating the performance of *BOSS* through pairing with an iPhone 7

public through Git. We envision that *BOSS* will receive the community's attention, and will contribute towards its code upon the release of a stable version. Also, we are developing an application to leverage the *BOSS* capabilities in various domains (e.g. wearable and vehicular).

CHAPTER 6

EDGE APPLICATIONS

In this chapter, we present three applications *BLINK*, *3D Story Teller*, and *DriveBlue* to utilize edge computing, with each representing a different category. *BLINK* depicts the merits of having an open source Bluetooth stack. In *3D Story Teller*, a single application splits its execution on more than one device. *DriveBlue* collects data from multiple devices and processes them on the edge to avoid misusing the bandwidth with raw data transmission. In the remainder of this chapter, we will explain the three applications and the lessons learned from each.

6.1 BLINK: MULTICASTING IN BLUETOOTH PICONETS

6.1.1 OVERVIEW

BLINK a customized Bluetooth v4.0 that enables multicast communication within a piconet. *BLINK* is built on top of *BOSS* to target IoT applications in need of low latency communication, e.g., drone fleet management. In order to support multicasting, we proposed a new packet design to address any subset of the piconet devices. Moreover, *BLINK* packets enable any applications to design their own set of instructions and features. Also, we proposed a modified Bluetooth time allocation scheme that allows multiple devices within a piconet to simultaneously exchange packets with no collision. To evaluate *BLINK*, we created a testbed of four devices, one master and three slaves, to exchange low latency packets in managing a fleet of drones. Our experiments showed that *BLINK* enhances the amount of data exchange between 1.5-3 folds compared to classical Bluetooth v4.0.
BLINK offers three main contributions:

1. Design *BLINK*, a customized version of Bluetooth v4.0 which enables multicast communication within a Bluetooth piconet. The design includes new packet format for multicast communication and modified time scheduling between master and slave devices.

2. Implementation and evaluation of *BLINK* in comparison to classical Bluetooth v4.0 and the ability to show data exchange enhancements more than 1.5 folds.

6.1.2 USE CASE

Intel is leading the way in drones with its projects *Drone 100*, *Drone 300*, and *Drone 500* [152]. These projects enable a group of drones to fly together to perform a certain mission, e.g., a light work display, as in Super Bowl 2017, or search and rescue. Upgrading from a 100 to a 500 drones fleet has been challenged, in part, by how to coordinate changes in flight parameters, e.g., altitude and speed, for the large number, especially with an inter-drone distance of 5ft. Complex algorithms have been used to calculate trajectories for all of the fleet and an Intel IoT Gateway per drone has enabled communicating the information from a cloud server [153]. However, expanding further is challenged in part by communication latency.

In this paper, we present *LAMEN* to target this category of applications in need of low latency interactions. *LAMEN* offers creating a groups of 8, the max piconet size, with a master in charge of cloud communication and command dissemination within its piconet. In managing a fleet of drones, we expect the close ones to be performing similar action; hence, organizing them in piconets sounds practical. We envision that this will cut the communication overhead by a factor of the piconet size. For example, in an 8 device piconet, 1 master and 7 slaves, the cloud communication will be reduced by a factor of 8, in theory, managing 800 drones as if they were 100.

6.1.3 DESIGN

We describe the design of *BLINK* including its multicast communication support, packet format, addressing modes, and instruction set.

Multicast Piconet Communication

BLINK, is built on top of *BOSS*, a customized Bluetooth version for IoT applications. *BLINK* targets applications in need of low latency communication through modifying the communication strategy within a piconet from unicast to multicast while maintaining other Bluetooth features. Currently, within a piconet, the master communicates with one slave at a time. For example, if the same message is sent to three slaves, it would be equivalent to exchanging three messages between

FIG. 27: Bluetooth v4.2 Data Channel PDU.

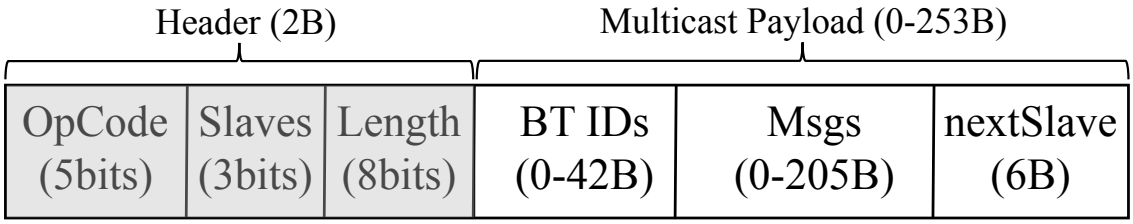


FIG. 28: Payload for *BOSS* Multicast Communication's data channel PDU.

a master and a slave. This clearly imposes communication latency, as shown later in § 6.1.4, that contradicts the rapid pace expected by IoT applications, e.g., drone flight management.

On the other hand, *BLINK* enables multicast communication through two phases control and data exchange. First, in the control phase, while establishing the piconet connections, the *master* assigns the same frequency hopping parameters to all of the slaves; hence, they all reside on the same channel at the same time. Second, in the data exchange phase, *BLINK* uses the payload in Bluetooth data packets, shown in FIG. 27, to send application-specific information, e.g., drone flight assistance, and collision avoidance.

Bluetooth piconets enable full duplex communication between master and slave. In case of downstream, a *master* transmitting to any number of slaves, no collision is expected. However, in the upstream, if two or more slaves attempt to respond in the same time slots on the same channel, messages will fail, due to packet collision. Hence, *BLINK* uses a round robin approach to listen to one slave at a time. Bluetooth assigns specific time slots for both master and slave to transmit in. Since all of the slaves are set on the same channel, whenever a *master* is transmitting it will announce the ID of the slave to use the upstream slot. This approach will limit the multicast communication to the downstream only, which is critical in enhancing the IoT communication latency.

Packet Format

The *BLINK* packet, shown in Fig 28, is hosted in the payload of the Bluetooth data packet, which enables up to 255B of payload in its packet as shown in Fig 27. *BLINK* splits this payload into two portions: two bytes of header and up to 253 bytes of multicast payload.

Header carries three pieces of information within its two bytes: OpCode, Slaves, and Length. *OpCode*, a 5 bits of Operation Code to define what this message has to offer. It enables defining up to 32 (2^5) different message types. *Slaves*, a 3 bit field to declare the number of slaves this message has to target. It supports addressing up to 7 devices as the maximum number of slaves within a piconet. *Length*, an 8 bit field to identify the length of the payload in bytes. Although it supports up to 255 bytes (2^8), it can only address up to 253 remaining in the Bluetooth data packet after consuming 2 bytes by the header.

Multicast Payload, carries the actual message transmitted within a piconet. It consists of two fields: BT_IDs and Msgs. *BT_IDs* or Bluetooth IDs carry the IDs of intended recipients of the message within a piconet. Excluding the transmitter, the piconet remains with up to 7 active devices eligible to receive the message; hence, the BT_IDs field holds up to 42 bytes, since every Bluetooth address consists of 6 bytes. This field is only used when addressing a subset of the piconet, however, if a broadcast is required this field remains empty and the *slave* field in the header is set to {000}. This increases bandwidth optimization through eliminating useless data transmission and minimizing the time taken by each receiver to process the packet checking its presence in the receivers list. The end of the *BT_IDs* field is identified through counting the number of slave bytes from the header. Also, this declares the beginning of *Msgs* fields, which carries the actual payload in *BLINK*. The end of *Msgs* is identified by the length field in the header taking away 6 bytes for the next field. *nextSlave*, a 6 bytes field used exclusively for masters to assign which slave will receive the token and transmit, which is used to prevent multiple slave packets from collision. Slaves set this field to 0, since they are not authorized to pick another slave.

Addressing Modes

Bluetooth support for unicast communication is controlled by source and destination addresses. In *BLINK*, for backward compatibility, we keep the destination field holding the first slave's ID and expand on it in the payload. This enables the sending of dedicated messages to a subset of the Bluetooth piconet; hence, it supports both multicast and broadcast. *Multicast support* is achieved through specifying a list of devices in *BLINK*'s packet to declare the set of eligible receivers. *Broadcast* is supported through setting the *slaves* field in *BLINK*'s header to {000} and leaving the *BT_IDs* field empty. In Broadcast, with the absence of *BT_IDs* other devices outside the piconet will receive and process the packet. This is solved by adding an extra check if the receivers list is blank, confirm the sender ID is the piconet master.

Instruction Set

In support of the drone flight management scenario, *BLINK* provides an initial set of instructions. In this section, *BLINK* introduces 5 instructions: maintainStatus, moveHorizontal, moveVertical, rotate, and changeColor. Each instruction has a unique 5 bit code to be used in the *OpCode* field of *BLINK*'s header. All *BLINK* instructions share the same Bluetooth header (2 bytes) and MIC (8 bytes) fields, shown in Fig. 27. In generating *BLINK*'s packet, the *header*, *BT_IDs*, and *nextSlave* are set by the master based on the instruction. In the *Msgs* field each instruction carries different information. *maintainStatus* comes with an empty *Msgs* field as its *OpCode* suffices for the intent, it is only needed to maintain a connection active between master and slave as required by Bluetooth v4.0. *moveHorizontal* and *moveVertical* both comes with a 2 byte signed integer to represent the required motions distance with a specific direction. A positive integer implies forward or up while a negative value implies backward of down. *rotate* comes with a 2 byte signed integer to represent the angle of rotation, with the positive value implying clockwise rotation. *changeColor* carries a 4 byte field to represent the new color the drone has to show. The 4 byte length was assigned by Intel's Drone 500 project to illuminate in 4 billion different color combinations.

The instruction list is not inclusive; it only aims to illustrate how to define a new instruction and include it the packet format. It also leaves room for adding more instructions as needed by the target application.

6.1.4 IMPLEMENTATION AND EVALUATION

In this section, we evaluate multicast communication in Bluetooth piconet, however, our ultimate goal is to show the value of having an open source Bluetooth stack. Using 4 Ubertooth One devices, we emulate a Bluetooth Piconet and show how data is exchanged between devices. Our key finding is that, despite extra control message exchange using multicast communication within a Bluetooth Piconet results in an overall larger data exchange per minute, between 1.5-3 folds, compared Bluetooth to 4.0 unicast piconet communication.

Experiment Setup

In evaluating *LAMEN* we connected 4 Ubertooth One devices to two laptops each running a different operating system, i.e., macOS and Ubuntu. All Ubertooth One devices run the same version of *BOSS* or *BLINK* based on the experiment mode.

The experiment begins by connecting Ubertooth One devices to the laptops and giving each a unique Bluetooth ID. Then, establishing a Bluetooth connection of two phases control and data exchange. The *control phase* begins by randomly choosing a device to be the master; the remaining devices initiate communication with slaves by sending ADV_IND packets, and the master replies to each with a CONNECT_REQ packet to establish a connection and to create a piconet. The *data exchange phase* begins with all devices synchronized and ready to exchange messages. In this section, we use the *data exchange phase* in three modes: unicast, multicast downstream, and multicast upstream. *Unicast*, with an unmodified version of *BOSS*, represents the Bluetooth 4.0 standard using unicast in series communication between the piconet slaves receiving messages from the master. The *multicast downstream* mode runs *BLINK* to provide multicast/broadcast communication enabling master to communicate with all slaves concurrently. The *multicast upstream* mode is similar to Multicast downstream, – it enables slave to exchange messages with master one at a time, to avoid packet collision. In the latter two modes, we use *LAMEN* packets, as shown in Fig. 28, while in the former mode we use the Bluetooth PDU, as shown in Fig. 27.

Experiment Scenario

In this experiment, we emulate a scenario in which multiple Bluetooth devices are exchanging messages, e.g., a fleet of drones passing flight management information.

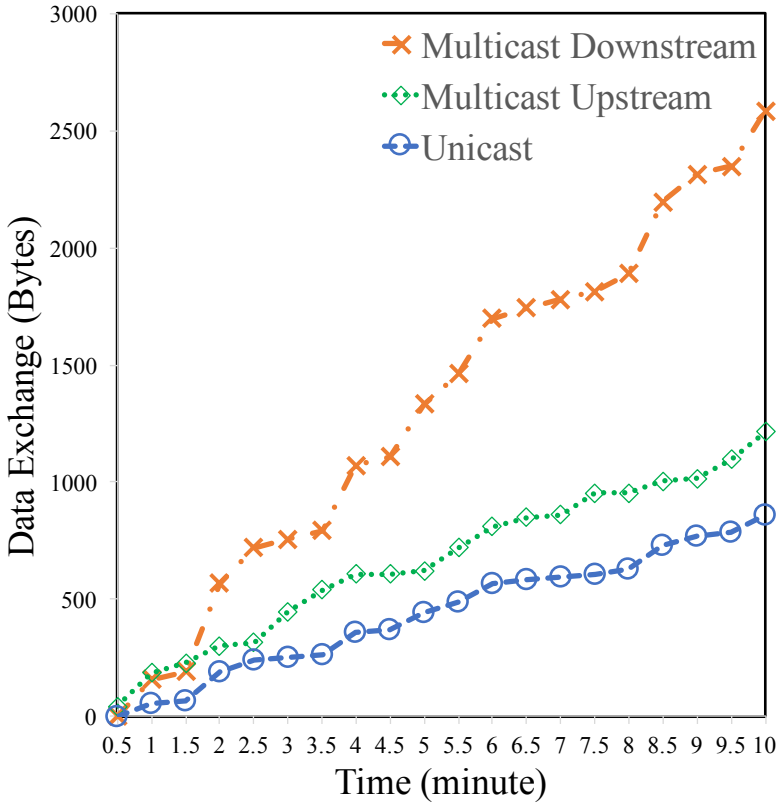


FIG. 29: Multicast Piconet Communication vs. Bluetooth default serial communication within a four device Piconet.

On the *master's* turn to transmit, we generate a random set of 2-15 instructions; unless the first one is the *maintainStatus* instruction. In this case, we send it alone. This emulates a master that informs the piconet members to adjust their coordinates or their illumination colors. Also, the master assigns a slave at a time to reply, based on a round-robin approach. Slaves respond to the master to report status, to report a problem, or to communicate messages from proximal piconets. The latter enables a multi-hop message between the piconets in order to coordinate between a large groups of devices.

Multicast Piconets Communication

In this experiment, we exchange messages between piconet devices for 10 minutes in three modes: *unicast*, which is the standard Bluetooth v4.0 in the form of *BOSS*; *multicast downstream*, which is *BLINK*'s master sending multicast messages to piconet slaves and not expecting responses; and *multicast upstream*, which is *BLINK*'s

master sending messages to piconet slaves and collecting their responses in series.

FIG. 29 depicts the behavior of all the three approaches. *Multicast downstream* has a clear advantage in terms overall data transmission within a piconet. It enables master communication with all of the slaves at the same time showing an overall data transfer of 2.8 folds more than *unicast* over the 10 minute experiment time. This increase in data exchange is almost a factor of the number of slaves within the piconet, showing multicasting to be efficient. However, when two-way communication between the master and the slaves is performed, the factor of data exchange dropped to ≈1.5, as shown in *multicast upstream*. This drop is a result of waiting for one slave at a time to respond, which is mandatory to avoid packet collision on the channel.

In real life, despite no way to evaluate on a large scale, we envision that not each slave will have a message to transmit at all time slots leaving the master more time slots to utilize. Hence, we anticipate that the overall enhancement will be somewhere between the multicast downstream and upstream data exchange rates.

6.1.5 CONCLUSION

In this application, we presented *BLINK*, an open source customized version of Bluetooth v4.0 to enable multicasting between devices within a Bluetooth Piconet. First, we implemented a full stack Bluetooth v4.0. Second, we proposed a new design for Bluetooth data packets to carry multicast communication. Third, we proposed a scheme to organize the transmission time slots between master and slave devices, in order to avoid packet collision. Fourth, on a real testbed, we evaluated *BLINK*, compared to classical Bluetooth v4.0, and showed an enhanced data exchange rate between 1.5-3 folds in a three slave piconet.

6.2 3D STORY TELLER

6.2.1 OVERVIEW

Streaming applications are widely available on smartphones and tablets. People use them to listen to news, stories, and movies; however, in a professional setting, sound systems have various techniques to enhance the listening experience of users. For example, *5.1 surround sound* creates a feeling that the sound is coming from 6 directions [154].

FIG. 30: *3D Story Teller* uses multiple speakers to emulate surrounded sounds experience

3D Story Teller is a streaming application that utilizes speakers from devices available in its proximity. Once the main device starts to play any track, it discovers available speakers in the proximity and splits the track characters on the available devices [34, 33]. This creates the *surround sound* experience with up to 6 devices available nearby.

6.2.2 IMPLEMENTATION

We implemented a prototype for *3D Story Teller* using Android SDK on Samsung Galaxy Note N7000 and Samsung S5 phones. Further, we generated audio tracks of 2 characters each and used Pure Data (PD) [155] for audio signal processing.

The implementation comes in three phases: initiation, discovery, and execution. First, a phone initiates playing a track. Second, *3D Story Teller* discovers and allocates speakers from phones in the proximity. Third, code orchestration with the new phone is done, in order to assign a character and start playing. Through out the execution, *3D Story Teller* faces incidents like having a device join or leave, each with a special handling.

Device Join, When a *3D Story Teller* supporting devices shows in the proximity, it immediately joins the cluster. The master then utilizes available resources as needed by the application, up to 5 phones besides itself to emulate the *5.1 surround sound* feature [154].

Device Leave, is detected through a communication failure within the cluster. If the device was executing a task, the master claims back the character and plays it until its claim for another speaker is granted of the track ends.

6.2.3 PERFORMANCE MEASURES

We monitored the execution of the smartphones, using Monsoon power [156], in three situations: executing the application alone in a *single* device scenario, *master* who initiates the application and assign roles to other devices, and *slave* who receives a character to play when available within the master's proximity.

Further, in order to investigate the communication overhead we monitored each device individually in the single situation in FIG. 32, in the master situation in FIG. 33, and in the slave situation in FIG. 34. The figures show the all situations operate within the 2000mW range except for the initial part in both *master* and *slave*. This suggests the communication overhead is imposed only during the communication initiation, while the single messages to start/stop execution on the slaves are insignificant.

6.2.4 CONCLUSION

In this application, we showed how devices could be orchestrated towards executing an application that was not possible using a single device. We showed that, using a small communication overhead we can utilize resources in the proximity without hampering the power consumption of edge devices, which is a major concern for all wearables, smartphones, and IoT devices in general. Although this attempt is preliminary, we can build more robust applications on top of it.

6.3 DRIVEBLUE: TRAFFIC INCIDENT PREDICTION THROUGH SINGLE SITE BLUETOOTH

6.3.1 OVERVIEW

In this application, we target processing large datasets at the edge of the network

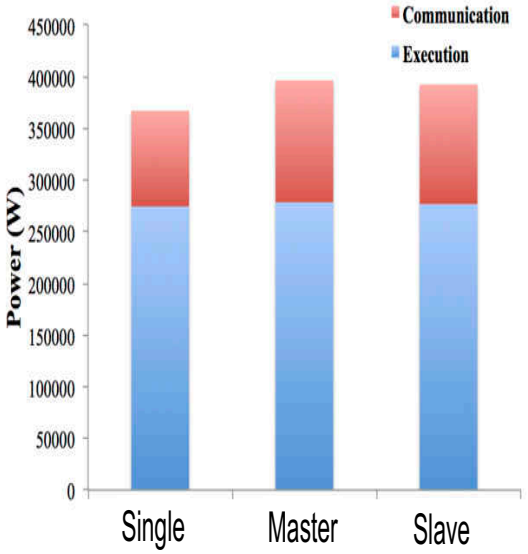


FIG. 31: Overall Power Consumption

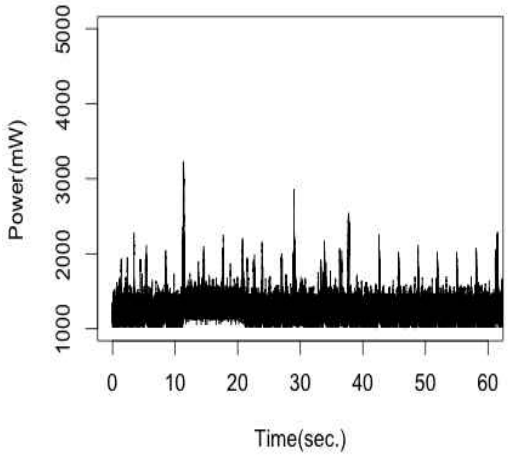


FIG. 32: Single Smartphone Power Consumption

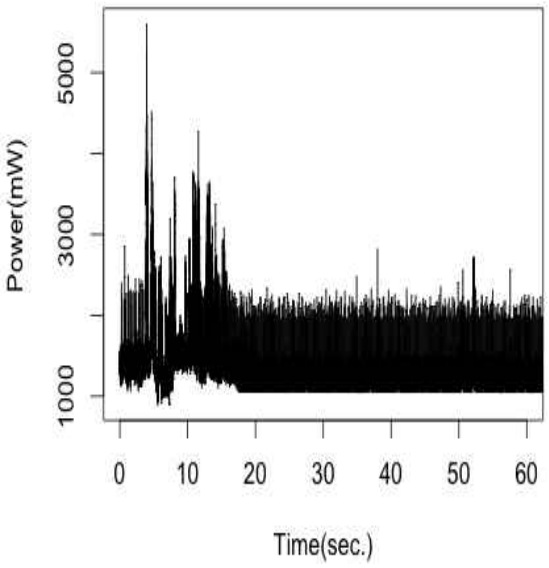


FIG. 33: Master Power Consumption

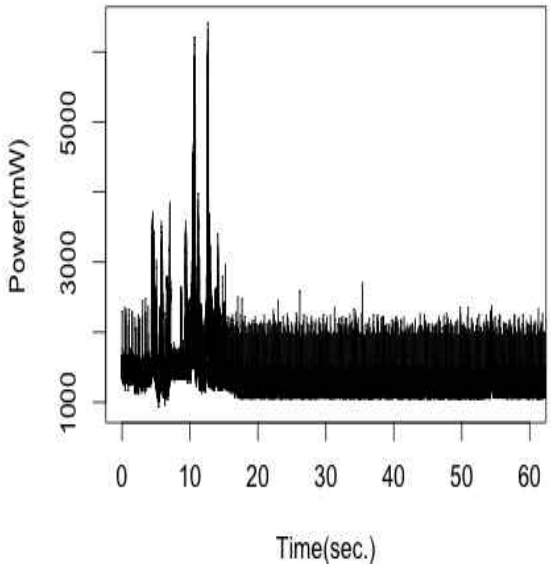


FIG. 34: Slave Power Consumption

in order to avoid bandwidth misuse in transmitting them to a cloud server for processing. *DriveBlue* works in the domain of intelligent transportation systems, which is getting much interest, with smart vehicles and smart city initiatives.

Traffic delays puts on 5.5 billion hours on the annual commute time, wasting 1.9 billion gallons of gas [157]. It has been estimated that every driver wastes approximately 34hrs/year, spending almost $713 because of traffic congestion. Two million new vehicles added to the US market annually drag these numbers worse year after year [158]. Traffic incidents (e.g. congestion, weather, and accidents) contribute by 45% to the annual congestion rates [159].

Real-time sensing is widely accepted with the evolution of smartphones, smart vehicles, and the Internet of Things (IoT). Smartphone usage doubled from 2010 to 2014, and is expected to double again in 2018 [160]. Smartphones loaded with sensors and above all available to commuters all the time can be easily utilized in the real-time sensing. Smart vehicles, e.g., Tesla, with all of their sensing, and communication capabilities, e.g., Bluetooth, are not far from the market. In 2013 they showed a 48% selling increase in California [157].

In this application, we propose *DriveBlue*: a single-site Bluetooth system to predict, analyze, and inform authorities of the current traffic conditions. Drivers' with Bluetooth devices (e.g., smartphones, the car's Bluetooth, and hands-free devices) turned on while driving, will enable *DriveBlue* to collect data expressing traffic conditions [143, 144]. *DriveBlue* places edge units each consisting of multiple Bluetooth adapters on a single site to compensate for the Bluetooth's randomness. Bluetooth units collect and process data on the edge of the network and report back to the authorities if they detect an incident. Traces collected from all vehicles on the road show an average of 10 devices per minute –which advocates for the use of Bluetooth in transportation applications.

Considering the highway case, we realized the presence of two types of lanes regular and High Occupancy Vehicle (HOV) lanes. The latter lanes are used to provide a carpooling incentive for extra speed. *DriveBlue* provides a module to separate data from those 2 lane types through machine learning techniques with an accuracy of $\approx 80\%$. Clearly, using the data without separation will give misleading results. For example, low traffic HOV lane readings can show a congested regular lane as a smooth traffic one. Moreover, *DriveBlue* provides a model to estimate traffic incidents from collected data, per lane type over time. Finally, *DriveBlue* uses

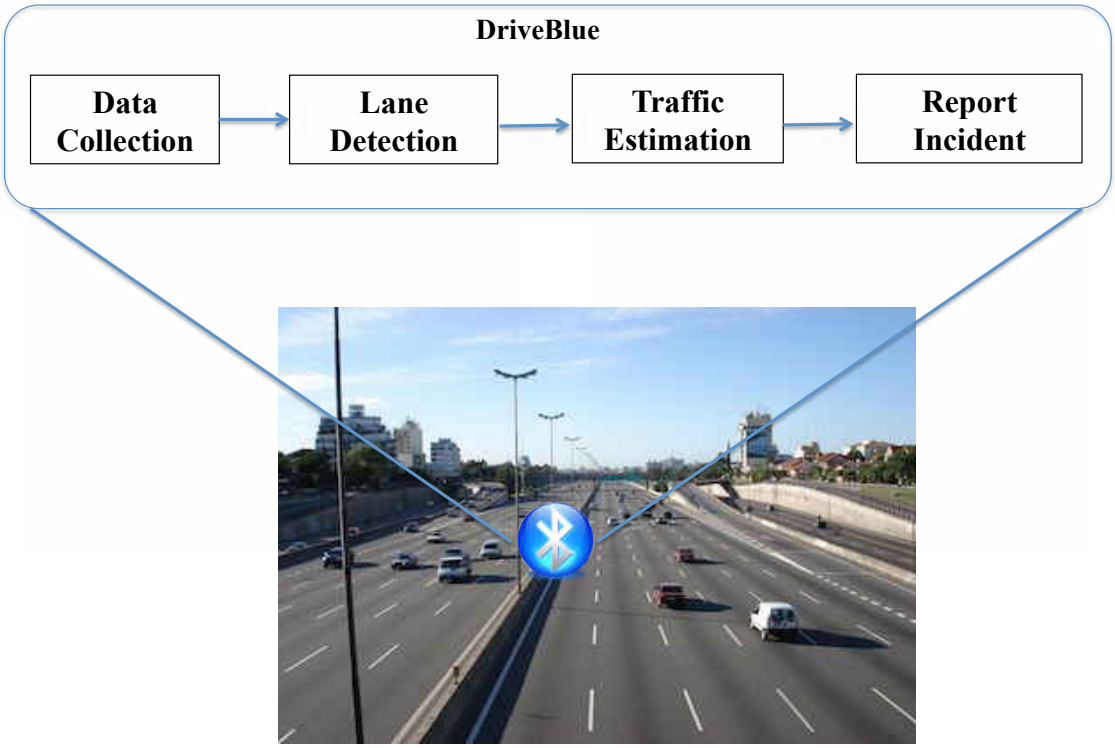


FIG. 35: *DriveBlue* System Design with the components of the Edge processing unit

LAMEN to process the data on the edge of the network to save the bandwidth for worthy usages, and to communicate back only a detected suspicious incidents. We evaluate the performance of *DriveBlue* using data collected from the highway to show its feasibility.

DriveBlue provides a traffic incident prediction system using single site Bluetooth adapters. *DriveBlue*, built on top of *LAMEN*, is used to process large data traces collected at the edge of the network instead of sending them for execution over the cloud.

6.3.2 DESIGN

In this application, we describe the design details of *DriveBlue*: a system to detect and report early symptoms of traffic incidents (e.g., traffic congestion), as shown in FIG. 35. First, we explain how to collect Bluetooth traces, and then lane

type detection, followed by how to predict upcoming traffic conditions, and finally how *DriveBlue* reports a situation that might escalate to a traffic jam.

Bluetooth Data Collection

Transportation authorities, e.g., VDOT, have Bluetooth adapters deployed on highways [161]. In this application, *DriveBlue* utilizes pre-installed infrastructure to initiate the discovery process for the Bluetooth data sample collection from devices passing by. The collected data provides the sender's device ID, timestamp, and received signal strength indicator (RSSI). The collected data is cleaned of noise, and outliers as explained later in section 6.3.3. The clean data is fed to the next module.

Lane Type Detection

It is critical to differentiate between various types of lanes on highways. This module uses the Support Vector Machine (SVM), a machine learning classifier to separate data received from regular and HOV lanes. SVM training phase uses k-fold cross validation to get the most out of the training data. The data used to train the classifier as follows: the *Bluetooth device ID* from the raw collected data; the *Appearance time* of the corresponding device is calculated by subtracting its initial appearance from the last; and *Mean, and Variance* of the RSSI values care ollected per device.

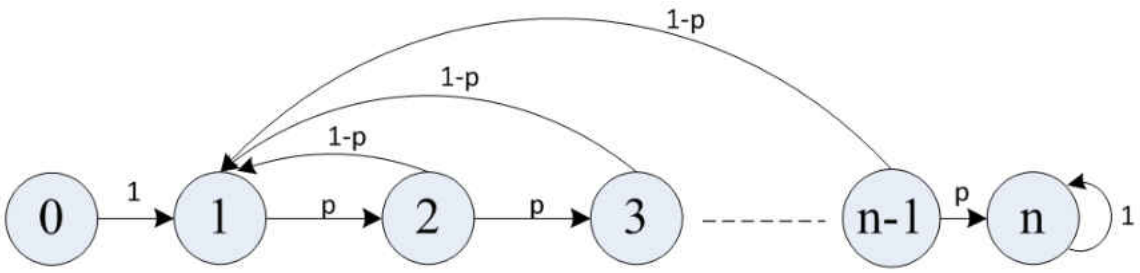


FIG. 36: Traffic Estimation Modeled as (n+1)-state Markov chain

Traffic Estimator

Upon receiving lane-specific data samples, this module begins to monitor the flow of traffic in the area of coverage. The module divides the time series into windows (e.g., 5 minutes) of data, each with the following values: number of devices recorded;

number of responses collected; mean, and variance of all of the RSSI values collected during the window. The traffic estimator declares a suspicious situation whenever three consecutive windows show higher-than-normal rates.

The traffic estimator is modeled as an (n+1) state Markov chain (T_u) shown in FIG. 36, where circles representing the traffic window marked an incident. The transition probability is denoted by $p_{x,y}$ where x, and y are the rows, and columns of the transition matrix. $p_{x,y}$ of the Markov chain are written as follows:

$$Pr\{T_{u+1} = y \mid T_u = x\} = \begin{cases} 1 & \text{y=x=n} \\ & \text{or y=1-x x<1,} \\ p & \text{y=x+1,} \\ 1-p & \text{y=1 0<x<n,} \\ 0 & \text{otherwise,} \end{cases}$$

$$\text{where } x, y = \{0, 1, 2, ...n\}.$$

To decrease the error of miss detection, n is chosen to be 4, as 3 consecutive windows detecting an incident are less likely to have an error less than 13%, assuming that probabilities of having an incident or not are equally likely.

Report Incident

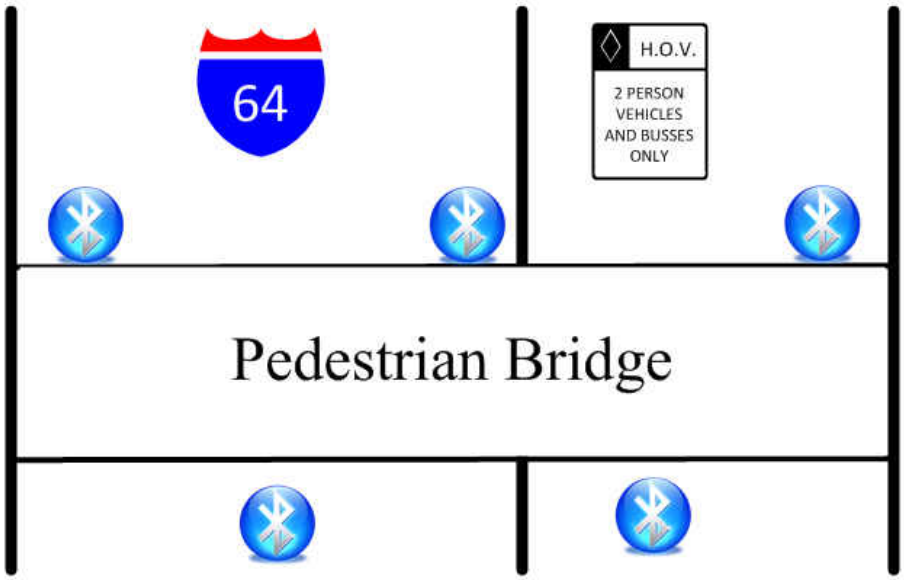
Vehicular communication introduced the use of Road Side Unit (RSU): infrastructure units on roads to bridge the gap between vehicles and an authority over the cloud [162].

In *DriveBlue*, we adopt the same architecture, in which Bluetooth units will have secure communication with the authorities. Upon notification of a suspicious situation from the traffic estimator, the reporter communicates it back to the authorities. It is now the authorities turn to validate, and to mitigate the upcoming situation.

6.3.3 PERFORMANCE EVALUATION

In this section, we evaluate the performance of *DriveBlue* through experiments. For logistic, and legislative conditions, we were allowed limited access to public roads for our data collection. Hence, the scope of our evaluation will not consider the traffic estimator module presented in § 4.4, since it requires larger dataset.

(a) Controlled



(b) Highway

FIG. 37: *DriveBlue* Field Experiments' Setup

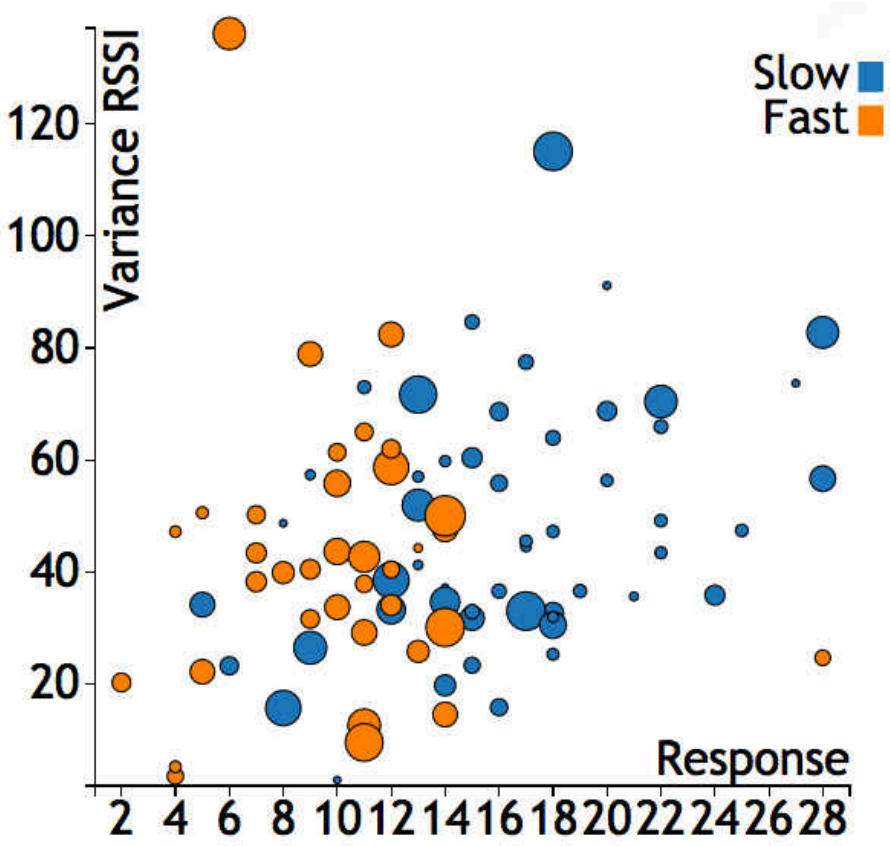


FIG. 38: Raw data from controlled experiment in *DriveBlue*

Experiment Setup

To evaluated *DriveBlue* we used: Android phones (i.e., Samsung Galaxy Note II and Nexus 4), Parani-UD100 industrial Bluetooth adapters [163], high gain antennas (i.e., 9dBi [164] and 15dBi [165]) to provide large coverage area ≈1600ft, and Lenovo Core i5 laptops running Ubuntu 12.04. In our experiments, we used multiple Bluetooth adapters at a single site to overcome the randomness of the discovery process. Also, we used two types of vehicles (i.e., SUV and full size) to consider the impact of different car design on Bluetooth signals.

We used two experiment setups as follows:

1. **Controlled Experiment**: The experiment was performed on Old Dominion University's (ODU's) campus to collect data from known devices. 6 Bluetooth receivers were placed on two sides of the road (3ft. width) as shown in FIG. 37a. To emulate multiple vehicles on the road, we loaded a vehicle with four Android

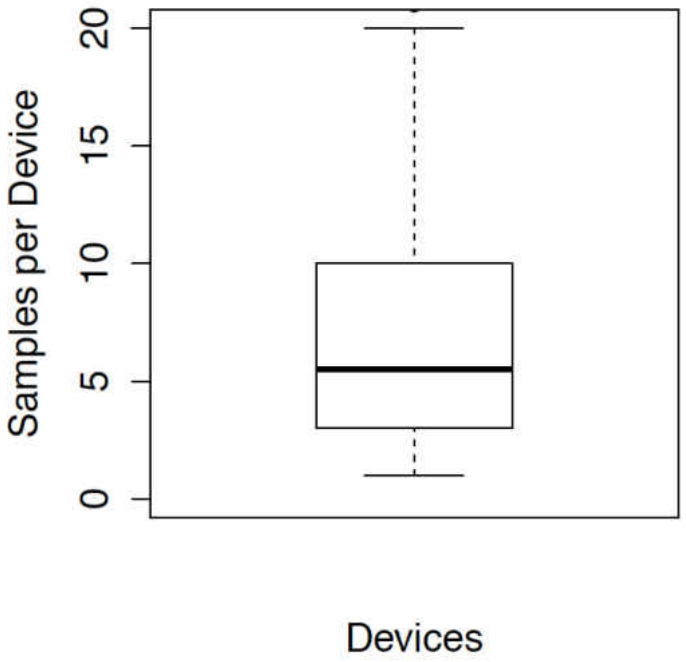


FIG. 39: Number of Samples Collected by *DriveBlue* per device from both lane types

devices with Bluetooth enabled. The vehicle spanned the antennas' coverage areas with two speed categories: fast (≥40mph), and slow (≤35). The experiment was performed at 12 runs per speed category (4 samples per run), and we collected a total of 96 samples. The collected samples were not linearly separable, as shown in FIG. 38; the bubble size represents the appearance time in seconds.

2. **Highway Experiment**: Upon receiving limited time allowance to perform our experiments (45 min.), 5 Bluetooth adapters were set on Virginia highway I-64, as shown in FIG. 37b. Two vehicles, each with 4 Android devices were used to commute through regular, and HOV lanes for 2 runs each.

Data Cleaning and Analysis

Controlled experiments collected 263 samples out of which 96 were used. On the other hand, Highway experiment collected 2230 samples from 300 different devices (promising number of devices considering the limited time). 1148 samples from 263 devices were used; 804 samples representing slow speed from 222 devices, and 344

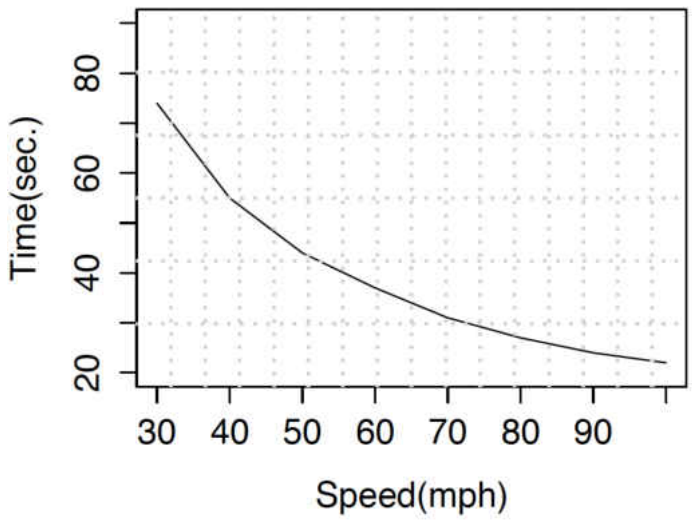


FIG. 40: Expected appearance time of Bluetooth devices on vehicles spanning the coverage area of a high gain antenna collected by *DriveBlue*

samples representing fast speed from 41 device.

Samples collected from both experiments were subject to the following: eliminating smartphone samples with only 1 appearance (0 Controlled, and 37 Highway), and eliminating samples sent by co-located Bluetooth adapters (117 controlled, and 232 Highway).

Data collected from highway experiment shows a tendency for a devices to provide more than 5 samples every time they span the adapter's coverage area regardless of the motion speed as shown in FIG. 39. Collected samples were subject to 2 kinds of analysis: data from known devices, and unknown source data. Due to the absence of ground truth in the later type, ambiguous data that fall between the ranges of our speed categories (35-40mph) were eliminated (813 samples from 62 different devices). FIG. 40 depicts the expected appearance time of Bluetooth devices loaded on a vehicles, based on the high gain antennas' coverage range.

Lane Type Detection

In *DriveBlue*, we map the speed categories to lane types, were slow represents driving in a regular lane, while fast represents driving on HOV lanes. Samples from controlled experiments were used to train two machine learning classifiers using 3-fold cross validation: SVM got an accuracy of 79%, and Logistic Regression (LR)

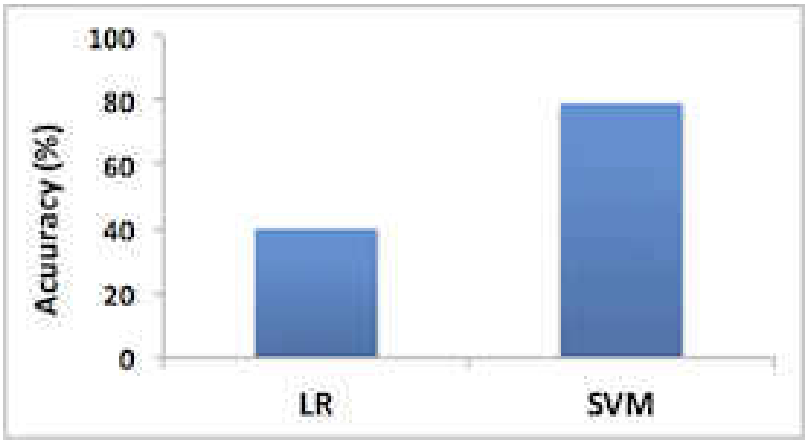


FIG. 41: Controlled Classifier Accuracy in Lane Type Detection

got 40% accuracy as in FIG. 41.

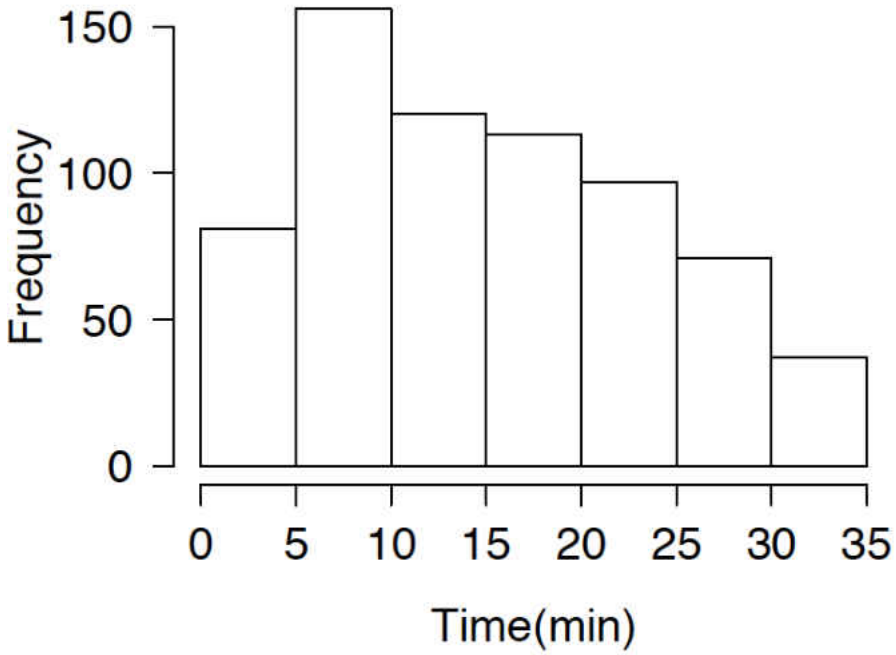
SVM shows an acceptable accuracy considering the limited set of training samples. The classifier was used to successfully detect the motion type of the known source samples used in the highway experiment as shown in FIG. 43. It is worth mentioning that, appearance time ,when used, showed enhancement in the classifier's accuracy. However, the main parameter that enhanced the accuracy was RSSI. This can be seen in FIG. 38, where the bubble size represents the appearance time, and shows no significant bubble size difference between the two motion categories.

SVM results advocate for the eligibility of single site Bluetooth for lane detection, especially if granted access to collect more data samples in realistic scenarios.

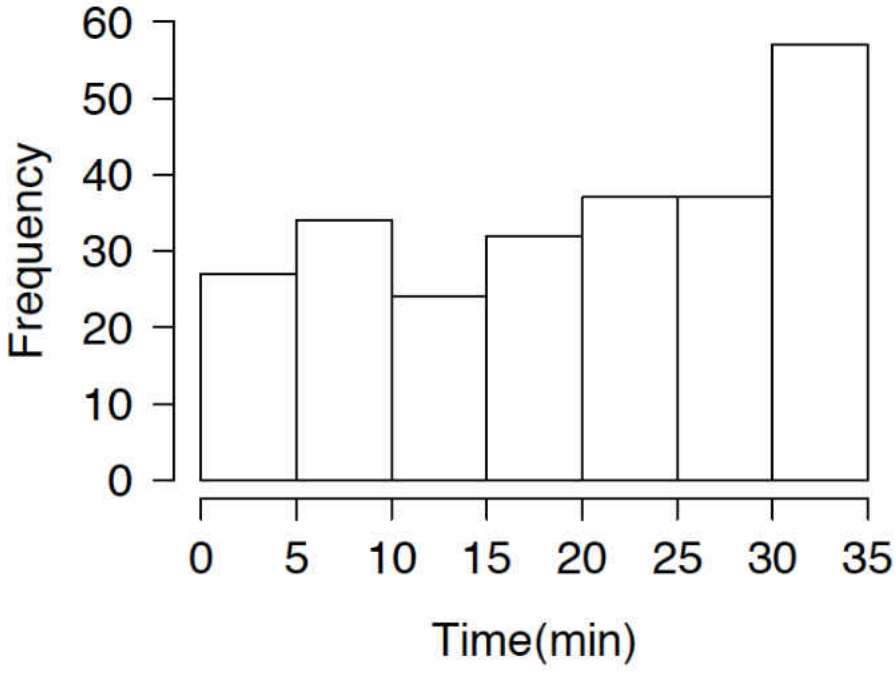
Highway Behavior Evaluation

The previous subsection, interpreted results from known source samples. However, this subsection considers unknown source samples. As mentioned earlier, due to the absence of ground truth and limited experiment time, samples that fall in the ambiguous range of the classifier were eliminated. Nevertheless, the remaining samples provided insights that prove the eligibility of single site Bluetooth, and shed light on future experiment setups.

Appearance Time: FIG. 42 depicts the difference of smartphone show ups between the slow and fast speed categories. The slow category collected an average of 134 samples per 5 minutes, while the fast category collected an average of 38 in the same range. Although, this finding needs to be validated with massive data collection



(a) Slow Speed



(b) Fast Speed

FIG. 42: Effect of Speed on *DriveBlue*'s samples Appearance Time collected in windows of 5 minutes each.

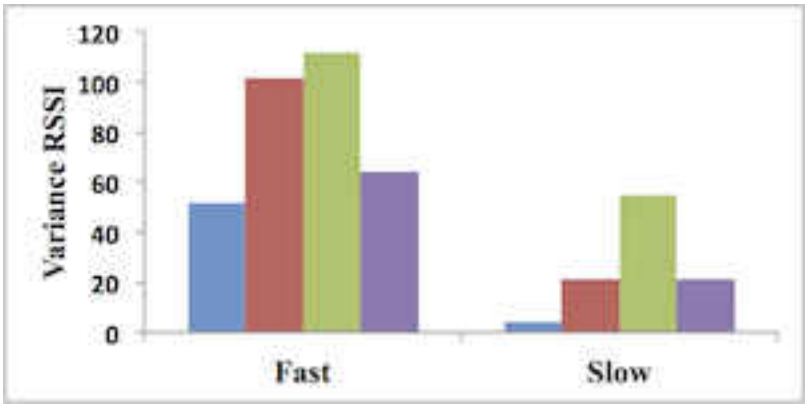


FIG. 43: *DriveBlue* Highway Readings, 4 smartphones color coded, for Lane Type Detection

at different times of the day, but the proposed approach can be used to generate a time series of expected appearance times per day. Continuous road monitoring can detect early gradual increase in these levels, and can report the likelihood of traffic jams in specific locations.

RSSI: FIG. 44 depicts the RSSI behavior spotted by slow, and fast motion per 5 minutes window. It is clear that slow motion shows almost steady reading levels; we envision this to change over larger time periods. Nevertheless, it provides the required average value that needs to be monitored for estimating future traffic congestion. Fast motion, shows variation of values due to the nature of HOV lanes where vehicles shows up less frequently than regular lanes. However, the average variation still shows a trend for the recorded period, which will be extended in the future work.

6.3.4 CONCLUSION

Data Collection: Data Collection needs to be extended on a wide scale of roads for longer times. Moreover, collecting data at different times of the day to account for congested, and free-flow traffic conditions. In addition to massive data collection, volunteers needs yo be recruited to provide a sufficient amount of ground truth. We envision that results from this work will encourage relaxed restrictions towards our data collection phase.

Direction Detection: Experiments were conducted in a location where traffic from both directions (e.g. north, and south) is significantly separated. This scenario was chosen to avoid Bluetooth traces from vehicles in the opposite direction as shown

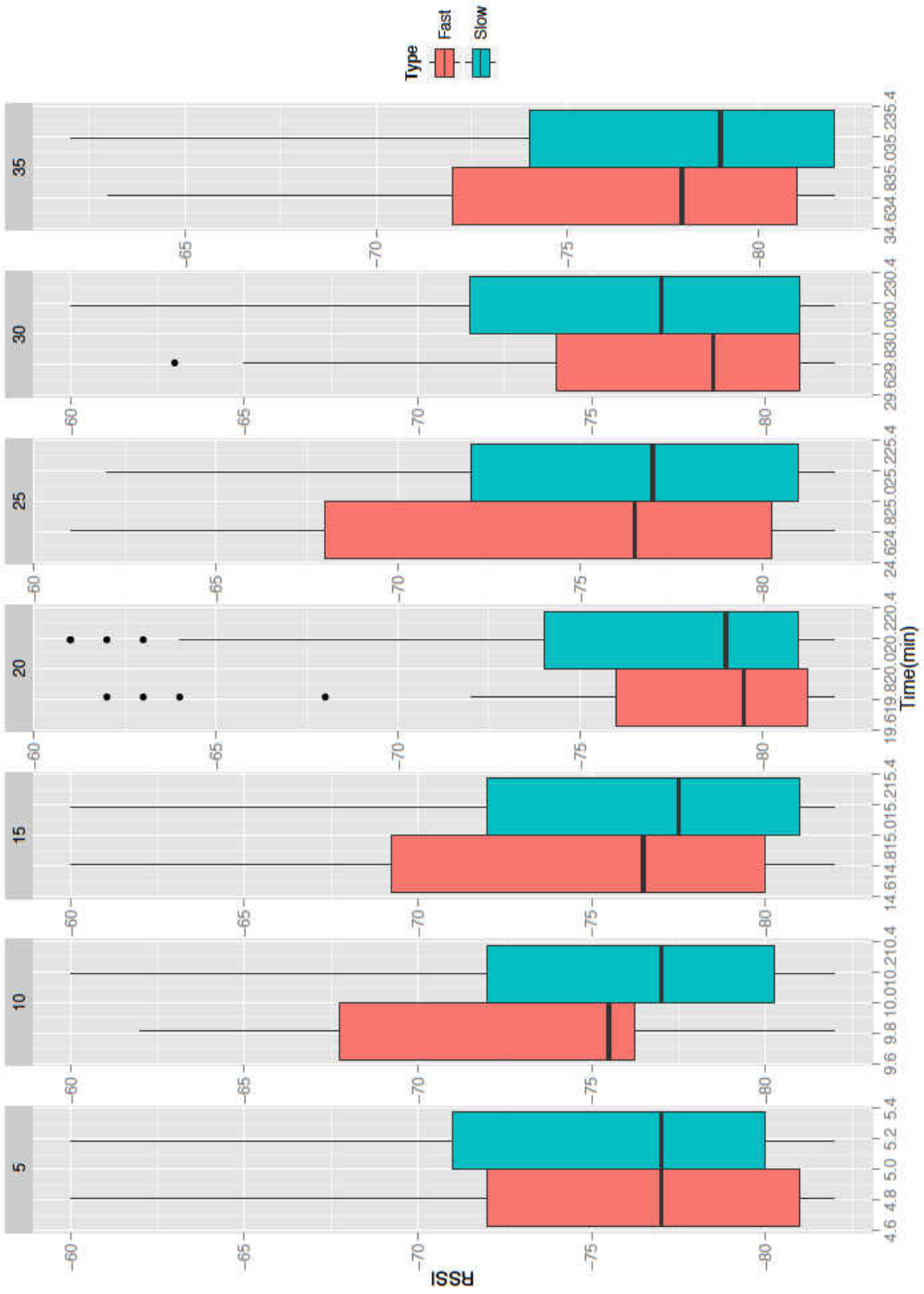


FIG. 44: *DriveBlue* RSSI collected on Highway. Each window represents 5 minutes comparison between motion categories

in FIG. 37b. Clearly, this is not the general case of highways. We envision direction classification to be done through the comparison of RSSI signals collected from different adapters on the same site, using a machine learning classifier.

Window Size: Traffic signals are divided into windows, which are compared together to monitor traffic over time. We envision the window size as an application specific parameter. For example, travel time estimation is less critical, and requires larger samples, than congestion prediction. Therefore, it is likely to separate applications by category,to define the features of each, and investigate the suitable window size for each of them.

In *DriveBlue*, we proposed an initial attempt for processing vehicular data at the edge of the network. *DriveBlue*, is a single site use of Bluetooth adapters for predicting traffic conditions. *DriveBlue* provided enough modularity to facilitate further modifications. First, we presented data collection and cleaning modules. Second, we showed how to differentiate between devices on regular and HOV lanes with ≈80% accuracy given the limited dataset. Third, we showed how to estimate the upcoming traffic condition. Fourth, we showed how to communicate our detected incident to the responsible authority. Finally, we used real data from highways to evaluate *DriveBlue's* performance.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation, we presented our design and implementation for a framework to leverage mobile resources at the edge of the network in executing applications beyond the capabilities of a single device. Our attempt was motivated in part by the large data wasted at the edge of the network, in addition to the availability of idle resources on smart devices. We showed that using this approach we not only use idle resources, but we enhance the bandwidth utilization through avoiding transmission of raw and redundant data. To achieve this goal, we reported three systems *LAMEN*, *Kinaara*, and *BOSS* that implement edge resource organization, discovery and allocation, and IoT communication, as explained earlier. Moreover, we reported three applications, *BLINK*, *3D Story Teller*, and *DriveBlue* to show a variety of directions in which edge computing can be applied. Finally, we shed light on the need for an incentive model to encourage using our systems and we identified the key metrics of the model.

In the future, we envision the role and demand for edge computing to increase, hence, we would like to point on some aspects that would help such growth. First, *Applications*. We showed a variety of applications that rely on edge computing. Their variation depends on how people define the edge, e.g., whether it is static or mobile. Applications have to be flexible to support both modes of edge computing. Second, *Scalability* within a cluster. We showed the scalability that we achieved, which works well for the applications in our scope. However, to go beyond that level requires a serious look into the clustering mechanism and into how multiple clusters can cooperate. Third *Mobility*. We have shown the impact of mobility of our edge computing approach. The overhead imposed came from two phases: finding a replacement and restarting execution on the new device. However, it would be worth studying whether a hand-off approach could be introduced to minimize the impact of restarting the application. Fourth *Resource Orchestration*. In our application, we have shown I/O resource orchestration; however, addressing other resources, i.e., memory and CPU, would require serious modifications on kernel level drivers.

CHAPTER 8

PUBLISHED WORK

8.1 CONFERENCE PROCEEDINGS

- **Ahmed Salem**, Nirmit Desai, Theodoros Salonidis, and Tamer Nadeem, "Kinaara: Distributed Discovery and Allocation of Mobile Edge Resources, " Proc. of the 13th IEEE Int'l Conference on Mobile Ad Hoc and Sensor Systems (MASS), 2017.

- **Ahmed Salem**, and Tamer Nadeem, "Exposing Bluetooth lower layers for IoT communication," Proc. of the 3rd IEEE World Forum on Internet of Things (WF-IoT), 2016.

- **Ahmed Salem**, and Tamer Nadeem, "LAMEN: Towards Orchestrating the Growing Intelligence on the Edge," Proc. of the 3rd IEEE World Forum on Internet of Things (WF-IoT), 2016.

- **Ahmed Salem** and Tamer Nadeem, "LAMEN: leveraging resources on anonymous mobile edge nodes," Proc. of the Eighth ACM Wireless of the Students, by the Students, and for the Students Workshop, 2016.

- **Ahmed Salem**, Tamer Nadeem, Mecit Cetin, and Samy El-Tawab, "DriveBlue: Traffic Incident Prediction through Single Site Bluetooth," Proc. of the 18th IEEE International Conference on Intelligent Transportation Systems (ITSC), 725-730, 2015.

- Mostafa Uddin, **Ahmed Salem**, Ilho Nam, and Tamer Nadeem, "Wearable Sensing Framework for Human Activity Monitoring," Proc. of the 2015 ACM workshop on Wearable Systems and Applications, 21-26, 2015.

- **Ahmed Salem**, Ayman Abdel-Hamid, and Mohamad Abou El-Nasr, "A dynamic key distribution protocol for PKI-based VANETs," Proc. of the IEEE IFIP Wireless Days (WD), 1-3, 2011.

8.2 JOURNAL

- **Ahmed Salem**, Ayman Abdel-Hamid, and Mohamad Abou El-Nasr, "The Case For Dynamic Key Distribution For PKI-Based VANETS," International Journal of Computer Networks & Communications 6, no. 1 (2014): 61.

8.3 POSTER/DEMO

- **Ahmed Salem**, Nirmit Desai, Theodoros Salonidis, and Tamer Nadeem, "Resource Hunting on the Edge," Proc. of IEEE/ACM Symposium on Edge Computing (SEC), 2016.
- **Ahmed Salem**, and Tamer Nadeem, "Poster: MU-MIMO throughput Enhancement for Enterprise Networks," Proc. of the 16th ACM International Workshop on Mobile Computing Systems and Applications, 2015.
- **Ahmed Salem**, and Tamer Nadeem, "Demo: ColPhone: a smartphone is just a piece of the puzzle," Proc. of the ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication,263-266, 2014.
- **Ahmed Salem**, and Tamer Nadeem, "Poster: ColPhone: a smartphone is just a piece of the puzzle," Proc. of the 20th annual Int'l conference on Mobile computing and networking, 417-420, 2014.
- **Ahmed Salem**, Tamer Nadeem, and Mecit Cetin, "Poster: DriveBlue: Can Bluetooth Enhance your Driving Experience?," Proc. of the 12th ACM annual Int'l conference on Mobile systems, applications, and services 382-382, 2014.

APPENDIX A

INCENTIVIZING USERS TO PARTICIPATION IN EDGE COMPUTING

Edge Computing and Crowd Sensing have many similar components. Both collect data on the edge of the network, however, edge computing performs processing close to the data sources instead of on the cloud servers. This similarity enables the reuse of similar components, e.g., incentive models, upon proper customization. In this chapter, we will explain two models of incentive estimation that can be adopted and we will show where they need customization towards edge computing.

A.1 INCENTIVE MODELS FOR CROWD SENSING

In crowd sensing we have two models for incentive calculation static price and bidding approach. In the static pricing model, resource/data requesters announce the price that they are willing to pay and resource owners opt-in if they accept the terms [63]. Bidding approaches comes into two flavors single and multiple biddings, meaning that a resource owner can bid to execute one or multiple tasks at a time [166, 167]. The bidding process begins with a set of tasks made available by cloud servers and resource owners begin to compete over them. A central unit is responsible for receiving the bids, processing them, and linking winners to tasks. Different approaches vary in terms of how they define bids or how they process and select winners.

A.2 INCENTIVES MODEL FOR EDGE COMPUTING

Applying the static price model in edge computing sounds like a straightforward method. *Mediators* maintain price rates per resource, monitor the performance, calculate resource usage per service, and distribute the reward over participating devices. In this approach price rates can be adjusted according to the cluster situation. For example, the rate for seizing an accelerometer available while having 10 units should be different from having a single unit within a cluster. Also, a user rated for

good results should differ than others. Moreover, resource requesting users should be enabled to bid or to limit their payments.

Bidding approaches require different handling in order to avoid an increased latency. In edge computing waiting for edge devices to bid, processing the bids, and declaring winners is a very lengthy process. It can be overridden, since all of the devices participating in *LAMEN* have already submitted their resources to the mediator. In *LAMEN*, we envision doing the bidding process between the cloud and the mediator layers; hence, all resources behind the mediator are virtually aggregated. In this customized approach, a mediator can place his bid for applications at the cloud that are ready for execution.

The details of how to design the bidding algorithm, what parameters are required, and how this impacts edge computing in terms of application latency and execution accuracy are left for future work.

REFERENCES

[1] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," *27th IEEE Int'l Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 775–787, 2013.

[2] Canalys, "Media alert: Over 1.5 billion smart phones to ship worldwide in 2016," https://www.canalys.com/newsroom/media-alert-over-15-billion-smart-phones-ship-worldwide-2016, Feb. 2016.

[3] Gartner, "Gartner Says 6.4 Billion Connected Things Will Be in Use in 2016, Up 30 Percent From 2015," http://www.gartner.com/newsroom/id/3165317, 2015.

[4] ABI research, "Data Captured by IoT Connections to Top 1.6 Zettabytes in 2020, As Analytics Evolve from Cloud to Edge," https://www.abiresearch.com/press/data-captured-by-iot-connections-to-top-16-zettaby/, 2015.

[5] EMC Digital Universe, "The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things," http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, April 2014.

[6] P. McGarry, "Why Edge Computing Is Here to Stay: Five Use Cases," https://www.rtinsights.com/why-edge-computing-is-here-to-stay-five-use-cases/, Oct. 2015.

[7] IT Business Edge, "How the Internet of Things Will Transform the Data Center," https://www.itbusinessedge.com/slideshows/how-the-internet-of-things-will-transform-the-data-center.html.

[8] C. McLellan, "The internet of things and big data: Unlocking the power," http://www.zdnet.com/article/the-internet-of-things-and-big-data-unlocking-the-power/, March 2015.

[9] Amazon AWS, "Amazon EC2 - Virtual Server Hosting," https://aws.amazon.com/ec2/.

[10] R. King, "Siri requires 100 times more computing power than text-based Web searches," http://www.biometricupdate.com/201504/siri-requires-100-times-more-computing-power-than-text-based-web-searches, April 2015.

[11] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 223–238, 2015.

[12] Y. Chen, S. He, F. Hou, Z. Shi, and J. Chen, "Promoting device-to-device communication in cellular networks by contract-based incentive mechanisms," *IEEE Network*, vol. 31, no. 3, pp. 14–20, 2017.

[13] Uber, "Uber," https://www.uber.com/.

[14] AirBnB, "Airbnb," https://www.airbnb.com/.

[15] AirPnP, "Airpnp," https://app.airpnp.co/.

[16] Karama, "Karma WiFi," https://yourkarma.com/.

[17] J. M. Greer, *Circles of Power: An Introduction to Hermetic Magic.* Red Wheel/Weiser, 2017.

[18] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," *Proc. of the Sixth Int'l Conference on Advances in Future Internet*, 2014.

[19] M. T. Beck, S. Feld, C. Linnhoff-Popien, and U. Pützschler, "Mobile edge computing," *Informatik-Spektrum*, vol. 39, no. 2, pp. 108–114, 2016.

[20] D. LeClair, "The Edge of Computing: It's Not All About the Cloud," http://insights.wired.com/profiles/blogs/the-edge-of-computing-it-s-not-all-about-the-cloud#axzz478jVsSBi, July 2014.

[21] S. Agarwal, M. Philipose, and P. Bahl, "Vision: the case for cellular small cells for cloudlets," *Proc. of the Fifth ACM Int'l Workshop on Mobile Cloud Computing & Services*, pp. 1–5, 2014.

[22] I. B. Mustafa, M. Uddin, and T. Nadeem, "Understanding the intermittent traffic pattern of http video streaming over wireless networks," *14th IEEE Int'l Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, pp. 1–8, 2016.

[23] M. Patel, Y. Hu, P. Hédé, J. Joubert, C. Thornton, B. Naughton, J. Joubert, C. Thornton, B. Naughton, N. Sprecher, T. Musiol, C. Manzanares, U. Rauschenbach, S. Abeta, L. Chen, K. Shimizu, A. Neal, P. Cosimini, A. Pollard, and G. Klas, "Mobile-Edge Computing," https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf, Sep. 2014.

[24] C. Wood, "Is Edge Computing Key to the Internet of Things?" http://www.govtech.com/transportation/Is-Edge-Computing-Key-to-the-Internet-of-Things.html, July 2015.

[25] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," 2015.

[26] R. Want, B. Schilit, and D. Laskowski, "Bluetooth le finds its niche," *IEEE Pervasive Computing*, no. 4, pp. 12–16, 2013.

[27] P. Kinney, "Zigbee technology: Wireless control that simply works," *Communications design conference*, vol. 2, pp. 1–7, 2003.

[28] R. Shorey and B. A. Miller, "The bluetooth technology: merits and limitations," *IEEE Int'l Conference on Personal Wireless Communications*, pp. 80–84, 2000.

[29] B. Cha, "Are voice commands on GPS worth it?" http://www.cnet.com/roadshow/news/are-voice-commands-on-gps-worth-it-ask-the-editors/, 2009.

[30] R. Amadeo, "Google to take on Nuance with speech recognition API," http://arstechnica.com/gadgets/2016/03/google-to-take-on-nuance-with-speech-recognition-api/, Mar. 2016.

[31] Oculus, "Oculus Rift," https://www.oculus.com/rift/.

[32] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "Crowddb: answering queries with crowdsourcing," *Proc. of the ACM SIGMOD Int'l Conference on Management of data*, pp. 61–72, 2011.

[33] A. Salem and T. Nadeem, "Colphone: a smartphone is just a piece of the puzzle," *Proc. of the 2014 ACM Int'l Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pp. 263–266, 2014.

[34] A. Salem and T. Nadeem, "Colphone: a smartphone is just a piece of the puzzle," *Proc. of the 20th ACM annual Int'l conference on Mobile computing and networking*, pp. 417–420, 2014.

[35] T. Okoshi, J. Ramos, H. Nozaki, J. Nakazawa, A. K. Dey, and H. Tokuda, "Reducing users' perceived mental effort due to interruptive notifications in multi-device mobile environments," *Proc. of the 2015 ACM Int'l Joint Conference on Pervasive and Ubiquitous Computing*, pp. 475–486, 2015.

[36] N. Zhang *et al.*, "Gameon: P2p gaming on public transport," *Proc. of the 13th ACM Annual Int'l Conference on Mobile Systems, Applications, and Services*, pp. 105–119, 2015.

[37] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," *Proc. of the 8th ACM Int'l conference on Mobile systems, applications, and services*, pp. 49–62, 2010.

[38] G. Sun and J. Shen, "Facilitating social collaboration in mobile cloud-based learning: a teamwork as a service (taas) approach," *IEEE Transactions on Learning Technologies*, vol. 7, no. 3, pp. 207–220, 2014.

[39] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[40] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[41] Amazon Web Services, "Amazon Web Services," https://aws.amazon.com/.

[42] Google, "Google Cloud Platform," https://cloud.google.com/.

[43] L. Alton, "The Pros and Cons of Cloud Computing," http://www.smallbusinesscomputing.com/biztools/the-pros-and-cons-of-cloud-computing.html, April 2015.

[44] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud computing: an overview." Springer, 2009, pp. 626–631.

[45] Level Cloud, "Advantages and Disadvantages of Cloud Computing," http://www.levelcloud.net/why-levelcloud/cloud-education-center/advantages-and-disadvantages-of-cloud-computing/.

[46] D. Reilly, C. Wren, and T. Berry, "Cloud computing: Pros and cons for computer forensic investigations," *Int'l Journal Multimedia and Image Processing (IJMIP)*, vol. 1, no. 1, pp. 26–34, 2011.

[47] Cloud Computing, "Cloud computing privacy concerns on our doorstep," *Communications of the ACM*, vol. 54, no. 1, pp. 36–38, 2011.

[48] H. Li, Y. Dai, L. Tian, and H. Yang, "Identity-based authentication for cloud computing." Springer, 2009, pp. 157–166.

[49] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE on Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[50] S. Ibrahim, H. Jin, B. Cheng, H. Cao, S. Wu, and L. Qi, "Cloudlet: towards mapreduce implementation on virtual machines," *Proc. of the 18th ACM Int'l symposium on High performance distributed computing*, pp. 65–66, 2009.

[51] M. Chen, Y. Hao, Y. Li, C.-F. Lai, and D. Wu, "On the computation offloading at ad hoc cloudlet: architecture and service modes," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 18–24, 2015.

[52] Y. Jararweh, F. Ababneh, A. Khreishah, and F. Dosari, "Scalable cloudlet-based mobile computing model," *Procedia Computer Science*, vol. 34, pp. 434–441, 2014.

[53] S. J. Stolfo, M. B. Salem, and A. D. Keromytis, "Fog computing: Mitigating insider data theft attacks in the cloud," *IEEE Symposium on Security and Privacy Workshops (SPW)*, pp. 125–128, 2012.

[54] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," *Proc. of the first edition of the ACM MCC workshop on Mobile cloud computing*, pp. 13–16, 2012.

[55] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics." Springer, 2014, pp. 169–186.

[56] A. Ahmed and A. Ejaz, "A survey on mobile edge computing," *Proc. of the 10th Int'l Conference on Intelligent Systems and Control (ISCO)*, 2016.

[57] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[58] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden, "Code in the air: simplifying sensing and coordination tasks on smartphones," *Proc. of the Twelfth Workshop on Mobile Computing Systems & Applications*, p. 4, 2012.

[59] F. Ye, R. Ganti, R. Dimaghani, K. Grueneberg, and S. Calo, "Meca: mobile edge capture and analysis middleware for social sensing applications," *Proc. of the 21st ACM Int'l conference companion on World Wide Web*, pp. 699–702, 2012.

[60] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84–106, 2013.

[61] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," *Proc. of 8th IEEE Int'l Conference on Cloud Computing (CLOUD)*, pp. 9–16.

[62] D. Chu, Z. Zhang, A. Wolman, and N. Lane, "Prime: a framework for co-located multi-device apps," *Proc. of the 2015 ACM Int'l Joint Conference on Pervasive and Ubiquitous Computing*, pp. 203–214.

[63] Amazon Mechanical Turk, "Amazon Mechanical Turk," http://tinyurl.com/hd346cs.

[64] J. Howe, "Crowdsourcing: A definition," *Crowdsourcing: Tracking the rise of the amateur*, 2006.

[65] J. Howe, "The rise of crowdsourcing," *Wired magazine*, vol. 14, no. 6, pp. 1–4, 2006.

[66] A. Kittur, E. H. Chi, and B. Suh, "Crowdsourcing user studies with mechanical turk," *Proc. of the ACM Special Interest Group on Computer-Human Interaction (SIGCHI) conference on human factors in computing systems*, pp. 453–456, 2008.

[67] H. Ma, D. Zhao, and P. Yuan, "Opportunities in mobile crowd sensing," *Proc. IEEE Communications Magazine*, vol. 52, no. 8, pp. 29–35, 2014.

[68] Y. Lee, Y. Ju, C. Min, S. Kang, I. Hwang, and J. Song, "Comon: cooperative ambience monitoring platform with continuity and benefit awareness," *Proc. of the 10th ACM Int'l conference on Mobile systems, applications, and services*, pp. 43–56, 2012.

[69] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt, "Micro-blog: sharing and querying content through mobile phones and social participation," *Proc. of the 6th ACM Int'l conference on Mobile systems, applications, and services*, pp. 174–186, 2008.

[70] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, "Rio: a system solution for sharing i/o between mobile systems," *Proc. of the 12th ACM annual Int'l conference on Mobile systems, applications, and services*, pp. 259–272, 2014.

[71] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan, "Medusa: A programming framework for crowd-sensing applications," *Proc. of the 10th ACM Int'l conference on Mobile systems, applications, and services*, pp. 337–350, 2012.

[72] M. Karaliopoulos, O. Telelis, and I. Koutsopoulos, "User recruitment for mobile crowdsensing over opportunistic networks," *Proc. IEEE Conference on Computer Communications (INFOCOM)*, pp. 2254–2262, 2015.

[73] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[74] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," *Proc. of the 1st ACM Int'l workshop on Wireless sensor networks and applications*, pp. 88–97, 2002.

[75] S. M. Puzi, S. Salleh, R. Ishak, and S. Olariu, "Neighborhood discovery in a wireless sensor networks," *Proc. of the 9th ACM Int'l Conference on Advances in Mobile Computing and Multimedia*, pp. 80–86, 2011.

[76] H. S. AbdelSalam and S. Olariu, "Toward adaptive sleep schedules for balancing energy consumption in wireless sensor networks," *IEEE Transactions on Computers*, vol. 61, no. 10, pp. 1443–1458, 2012.

[77] M. San Martın, C. Gutierrez, and P. T. Wood, "Snql: A social networks query and transformation language," *cities*, vol. 5, p. r5, 2011.

[78] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Transactions on database systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[79] M. Ruffing, Y. He, M. Kelly, J. O. Hallstrom, S. Olariu, and M. C. Weigle, "A retasking framework for wireless sensor networks," *Proc. IEEE Military Communications Conference (MILCOM)*, pp. 1066–1071, 2014.

[80] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communications magazine*, vol. 40, no. 8, pp. 102–114, 2002.

[81] H. S. AbdelSalam and S. Olariu, "Energy-efficient task management," *The Art of Wireless Sensor Networks*, pp. 385–425, 2014.

[82] H. S. AbdelSalam and S. Olariu, "Bees: Bioinspired backbone selection in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 44–51, 2012.

[83] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie, "Protocols for self-organization of a wireless sensor network," *IEEE personal communications*, vol. 7, no. 5, pp. 16–27, 2000.

[84] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. Srivastava, "Optimizing sensor networks in the energy-latency-density design space," *IEEE Transactions on Mobile Computing*, vol. 1, no. 1, pp. 70–80, 2002.

[85] I. Eyal, I. Keidar, and R. Rom, "LiMoSense: live monitoring in dynamic sensor networks," *Distributed Computing*, vol. 27, no. 5, pp. 313–328, 2014.

[86] S. Tilak, K. Chiu, N. B. Abu-Ghazaleh, and T. Fountain, "Dynamic resource discovery for sensor networks," *Embedded and Ubiquitous Computing–EUC 2005 Workshops*, pp. 785–796, 2005.

[87] X. Wang, A. Walden, M. C. Weigle, and S. Olariu, "Strategies for sensor data aggregation in support of emergency response," *Proc. IEEE Military Communications Conference (MILCOM)*, pp. 1120–1126, 2014.

[88] G. Fox, "Peer-to-peer networks," *Computing in Science & Engineering*, vol. 3, no. 3, pp. 75–77, 2001.

[89] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[90] P. K. Gummadi, S. Saroiu, and S. D. Gribble, "A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 1, pp. 82–82, 2002.

[91] W. You, B. Mathieu, P. Truong, J.-F. Peltier, and G. Simon, "Dipit: A distributed bloom-filter based pit table for ccn nodes," *Proc. of the 21st IEEE Int'l Conference on Computer Communications and Networks (ICCCN)*, pp. 1–7, 2012.

[92] Y.-S. Shim, Y.-S. Kim, and K.-H. Lee, "A mobility-based clustering and discovery of web services in mobile ad-hoc networks," *Proc. of the IEEE Int'l Conference on Web Services. ICWS 2009*, pp. 374–380, 2009.

[93] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[94] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Middleware 2001*, pp. 329–350, 2001.

[95] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on selected areas in communications*, vol. 22, no. 1, pp. 41–53, 2004.

[96] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[97] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, "Prefix hash tree: An indexing data structure over distributed hash tables," *Proc. of the 23rd ACM symposium on principles of distributed computing*, vol. 37, 2004.

[98] R. da Silva Villaca, L. B. de Paula, R. Pasquini, and M. F. Magalhães, "Hamming dht: Taming the similarity search," *IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 7–12, 2013.

[99] M. Rabinovich, Z. Xiao, and A. Aggarwal, "Computing on the edge: A platform for replicating internet applications," *Web content caching and distribution*, pp. 57–77, 2004.

[100] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multitenancy at the network's extreme edge," *IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 1–13, 2016.

[101] Open Connectivity Foundation, "AllJoyn Open Source Project," https://openconnectivity.org/developer/reference-implementation/alljoyn.

[102] Open Connectivity Foundation, "AllJoyn Members," https://openconnectivity.org/foundation/our-partners.

[103] Panasonic, "Wireless Speaker System SC-ALL3," http://www.panasonic.com/uk/consumer/home-entertainment/wireless-speaker-systems/sc-all3.html.

[104] LinkSys, "LinkSys EA8500 Max Stream AC2600 MU-MIMO Smart Wi-Fi Router," http://www.linksys.com/us/p/P-EA8500/.

[105] Open Connectivity Foundation, "The OCF Internet of Things Industries," https://openconnectivity.org/business/markets.

[106] Open Connectivity Foundation, "Iotivity, finding a resource," https://www.iotivity.org/documentation/linux/programmers-guide/finding-resource.

[107] S. Bluetooth, "The bluetooth core specification, v4. 0," *Bluetooth SIG: San Jose, CA, USA*, 2010.

[108] M. Krasnyansky, "Bluez: Official linux bluetooth protocol stack," http://www.bluez.org/, 2003.

[109] Android, "Android bluetooth, bluedroid," https://source.android.com/devices/bluetooth/.

[110] BlueSoleil, "Bluesoleil 10," http://www.bluesoleil.com/.

[111] Texus Instruments, "PAN1323 Bluetooth Evaluation Module Kit," http://www.ti.com/tool/ez430-rf256x, May 2014.

[112] Texus Instruments, "EZ430-RF256x Bluetooth Evaluation Tool," http://www.ti.com/tool/ez430-rf256x, May 2014.

[113] Texus Instruments, "MindTree EtherMind Bluetooth® Stack and SDK," http://www.embeddeddeveloper.com/tools/2319/Texas-Instruments/MT-BT-SDK.htm.

[114] Stephan, H. Fargus, J. Will, and Florian, "Retropi," https://retropie.org.uk/.

[115] Apache Mynewt, "Mynewt ble introduction," https://mynewt.apache.org/latest/network/ble/ble_intro/.

[116] A. Salem and T. Nadeem, "LAMEN: leveraging resources on anonymous mobile edge nodes," *Proc. of the Eighth ACM Wireless of the Students, by the Students, and for the Students Workshop*, pp. 15–17, 2016.

[117] A. Salem and T. Nadeem, "LAMEN: Towards orchestrating the growing intelligence on the edge," *3rd IEEE World Forum on Internet of Things (WF-IoT)*, pp. 508–513, 2016.

[118] V. Raychoudhury, J. Cao, R. Niyogi, W. Wu, and Y. Lai, "Top k-leader election in mobile ad hoc networks," *Pervasive and Mobile Computing*, vol. 13, pp. 181–202, 2014.

[119] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," *IEEE Symposium on Security and Privacy (SP)*, pp. 839–858, 2016.

[120] J. Hildenbrand, "What is sideloading?" http://www.androidcentral.com/what-sideloading-android-z, Feb. 2012.

[121] F. Baccelli, N. Khude, R. Laroia, J. Li, T. Richardson, S. Shakkottai, S. Tavildar, and X. Wu, "On the design of device-to-device autonomous discovery," *Proc. of the Fourth IEEE Int'l Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–9, 2012.

[122] A. Salem, N. Desai, T. Salonidis, and T. Nadeem, "Resource hunting on the edge," *Proc. of the IEEE/ACM Symposium Edge Computing (SEC)*, pp. 83–84, 2016.

[123] A. Salem, T. Salonidis, N. Desai, and T. Nadeem, "Kinaara: Distributed discovery and allocation of mobile edge resources," *Proc. of the 13th IEEE Int'l Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2017.

[124] Q. Ning, C.-A. Chen, R. Stoleru, and C. Chen, "Mobile storm: Distributed real-time stream processing for mobile clouds," *Proc. of the IEEE 4th IEEE Int'l Conference on Cloud Networking (CloudNet)*, pp. 139–145, 2015.

[125] H. Wang and L.-S. Peh, "Mobistreams: A reliable distributed stream processing system for mobile devices," *Proc. of the 28th IEEE Int'l Parallel and Distributed Processing Symposium*, 2014.

[126] N. J. Navimipour, A. M. Rahmani, A. H. Navin, and M. Hosseinzadeh, "Resource discovery mechanisms in grid systems: A survey," *Journal of Network and Computer Applications*, vol. 41, pp. 389–410, 2014.

[127] S. K. Datta, R. P. F. Da Costa, and C. Bonnet, "Resource discovery in internet of things: Current trends and future standardization aspects," *Proc. of the 2nd IEEE World Forum on Internet of Things (WF-IoT)*, pp. 542–547, 2015.

[128] Y. Fathy, P. Barnaghi, S. Enshaeifar, and R. Tafazolli, "A Distributed In-network Indexing Mechanism for the Internet of Things," *Proc. of the 3rd IEEE World Forum on Internet of Things*, 2016.

[129] E. Carlini, A. Lulli, and L. Ricci, "Dragon: Multidimensional range queries on distributed aggregation trees," *Future Generation Computer Systems*, vol. 55, pp. 101–115, 2016.

[130] H. Zhang, Y. Wen, H. Xie, and N. Yu, *Distributed hash table: Theory, platforms and applications*. Springer, 2013.

[131] T. Li, X. Zhou, K. Brandstatter, and I. Raicu, "Distributed key-value store on hpc and cloud systems," *2nd Greater Chicago Area System Research Workshop (GCASR).*, pp. 775–787, 2013.

[132] Kevik, "Camera in Android, how to get best size, preview size, picture size, view size, image distorted?" https://tinyurl.com/jfxl4su.

[133] Audacity Team, "Audio sampling rates," https://tinyurl.com/q7x9msm.

[134] Android Developers Guide, "Introduction to Android, Resource features," http://developer.android.com/guide/index.html.

[135] M. Lenczner and A. G. Hoen, "CRAWDAD ilesansfil/wifidog (v. 2015-11-06)," http://crawdad.org/ilesansfil/wifidog/20151106/session, Nov. 2015, traceset: session.

[136] M. Norouzi, D. J. Fleet, and R. R. Salakhutdinov, "Hamming distance metric learning," *Advances in neural information processing systems*, pp. 1061–1069, 2012.

[137] D. Lin *et al.*, "An information-theoretic definition of similarity." *International Conference on Machine Learning (ICML)*, vol. 98, no. 1998, pp. 296–304, 1998.

[138] Y. Sun, L. Ma, and S. Wang, "A comparative evaluation of string similarity metrics for ontology alignment," *Journal of Information & Computational Science*, vol. 12, no. 3, pp. 957–964, 2015.

[139] A. Salem and T. Nadeem, "Exposing bluetooth lower layers for iot communication," *Proc. of the 3rd IEEE World Forum on Internet of Things (WF-IoT)*, pp. 147–152, 2016.

[140] M. Ossmann, "Project ubertooth: Building a better bluetooth adapter," https://github.com/greatscottgadgets/ubertooth/wiki/Ubertooth-One, 2011.

[141] M. Uddin, A. Salem, I. Nam, and T. Nadeem, "Wearable sensing framework for human activity monitoring," *Proc. of the 2015 ACM workshop on Wearable Systems and Applications*, pp. 21–26, 2015.

[142] D. Hughes, "Ble overview," http://www.summitdata.com/blog/ble-overview/, 2013.

[143] A. Salem, T. Nadeem, and M. Cetin, "Driveblue: can bluetooth enhance your driving experience?" *Proc. of the 12th ACM annual Int'l conference on Mobile systems, applications, and services*, pp. 382–382, 2014.

[144] A. Salem, T. Nadeem, M. Cetin, and S. El-Tawab, "Driveblue: Traffic incident prediction through single site bluetooth," *18th IEEE Int'l Conference on Intelligent Transportation Systems (ITSC)*, pp. 725–730, 2015.

[145] M. Ettus, "USRP User's and Developer's Guide," *Ettus Research LLC*, 2005.

[146] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal, "Warp, a unified wireless network testbed for education and research," *IEEE Int'l Conference on Microelectronic Systems Education (MSE'07)*, pp. 53–54, 2007.

[147] Q. Wang and D. Agrawal, "UCBT - Bluetooth Extension for NS-2," http://www.cs.uc.edu/~cdmc/ucbt/.

[148] G. Tan, "Blueware: Bluetooth Simulator for NS-2," http://tinyurl.com/lksdg39.

[149] E. Blossom, "Gnu radio: Tools for exploring the radio frequency spectrum," *Linux J.*, vol. 2004, no. 122, Jun. 2004.

[150] Ellisys, "Ellisys bluetooth explorer, all-in-one bluetooth® analysis system," http://www.ellisys.com/products/bex400/.

[151] Fitbit, "Fitbit," https://www.fitbit.com.

[152] Intel, "Intel Drone Light Up the Sky," https://www.intel.com/content/www/us/en/technology-innovation/aerial-technology-light-show.html.

[153] Intel, "Intel IoT Gateway Technology," https://www.intel.com/content/www/us/en/embedded/solutions/iot-gateway/overview.html.

[154] X. Bosun, "Signal mixing for a 5.1-channel surround sound system'analysis and experiment," *Journal of the Audio Engineering Society*, vol. 49, no. 4, pp. 263–274, 2001.

[155] Pure Data, "Pure Data," http://puredata.info/.

[156] Monsoon Solutions Inc., "Monsoon Power Monitor." http://www.msoon.com/.

[157] Hedges and Company, "United states vehicle registration data, automobile statistics and trends," http://hedgescompany.com/automotive-market-research-statistics/auto-mailing-lists-and-marketing, 2015.

[158] D. Schrank, E. Bill, and L. Tim, "Ttiís 2012 urban mobility report," *Texas A&M Transportation Institute. The Texas A&M University System*, 2012.

[159] NationWide, "The real cost of traffic jams," http://www.nationwide.com/road-congestion-infographic.jsp.

[160] Statista, "Forecast: number of smartphone users in the u.s. 2010-2018," http://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/, 2017.

[161] Federal Highway Administration, "Traffic congestion and reliability: Trends and advanced strategies for congestion mitigation," https://ops.fhwa.dot.gov/congestion_report/, Sep. 2005.

[162] A. Hesham, A. Abdel-Hamid, and M. A. El-Nasr, "A dynamic key distribution protocol for pki-based vanets," *Proc. of the IEEE IFIP Wireless Days (WD)*, pp. 1–3, 2011.

[163] SENA, "Parani-ud100 bluetooth 4.0 class1 usb adapter, exchangeable antenna," http://www.senanetworks.com/?utm_source=tr.im&utm_medium=no_referer&utm_campaign=tr.im%2FXiPIx&utm_content=direct_input.

[164] Hawking Technology, "9dbi antenna," https://tr.im/WBJJG.

[165] Hawking Technology, "15dbi antenna," https://hawkingtech.com/product/hai15sc/.

[166] M. Li, J. Lin, D. Yang, G. Xue, and J. Tang, "Quac: Quality-aware contract-based incentive mechanisms for crowdsensing," *Proc. of the 14th IEEE Int'l Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2017.

[167] J. Xu, Z. Rao, L. Xu, D. Yang, and T. Li, "Mobile crowd sensing via online communities: Incentive mechanisms for multiple cooperative tasks," *Proc. of the 14th IEEE Int'l Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2017.

VITA

Ahmed Salem
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

EDUCATION

PhD. in Computer Science, Old Dominion University, USA, 2018.

M.S. in Computer Engineering, Arab Academy For Science and Technology, Egypt 2012.

B.S. in Computer Engineering, Arab Academy For Science and Technology, Egypt 2006.

PROFESSIONAL EXPERIENCE

2018 - **** Software Engineer, Amazon AWS Inc., USA.

2015 - 2015 Research Intern, IBM T.J. Watson Research Center, USA.

2013 - 2018 Graduate Research Assistant, Old Dominion University, USA.

2012 - 2013 Teaching Assistant, Old Dominion University, USA.

2011 - 2012 Software Engineer, POET A.G., Egypt.

2007 - 2011 Software Engineer, Integrated Solution For Ports (ISFP), Egypt.

PROFESSIONAL SOCIETIES

Association for Computing Machinery (ACM)

Institute of Electrical and Electronics Engineers (IEEE)

The ACM Special Interest Group on Mobility of Systems, Users, Data, and Computing (SIGMOBILE)

Typeset using LaTeX.