Promoting Systematic Practices for Designing and Developing Edge Computing Applications via Middleware Abstractions and Performance Estimation

Breno Dantas Cruz

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Eli Tilevich, Chair
Donald S. McCrickard
Na Meng
Rafael Matilla Montalvo
Francisco J. Servant Cortes

March 24, 2021
Blacksburg, Virginia

Keywords: Edge Computing, Programming Model, Data Sharing, Android, Performance Estimation, Machine Learning

Promoting Systematic Practices for Designing and Developing Edge Computing Applications via Middleware Abstractions and Performance Estimation

Breno Dantas Cruz

(ABSTRACT)

Mobile, IoT, and wearable devices have been transitioning from passive consumers to active generators of massive amounts of user-generated data. Edge-based processing eliminates network bottlenecks and improves data privacy. However, developing edge applications remains hard, with developers often have to employ ad-hoc software development practices to meet their requirements. By doing so, developers introduce low-level and hard-to-maintain code to the codebase, which is error-prone, expensive to maintain, and vulnerable in terms of security. The thesis of this research is that modular middleware abstractions, exemplar use cases, and ML-based performance estimation can make the design and development of edge applications more systematic. To prove this thesis, this dissertation comprises of three research thrusts: (1) understand the characteristics of edge-based applications, in terms of their runtime, architecture, and performance; (2) provide exemplary use cases to support the development of edge-based application; (3) innovate in the realm of middleware to address the unique challenges of edge-based data transfer and processing. We provide programming support and performance estimation methodologies to help edge-based application developers improve their software development practices. This dissertation is based on three conference papers, presented at MOBILESoft 2018, VTC 2020, and IEEE SMDS 2020.

Promoting Systematic Practices for Designing and Developing Edge Computing Applications via Middleware Abstractions and Performance Estimation

Breno Dantas Cruz

(GENERAL AUDIENCE ABSTRACT)

Mobile, IoT, and wearable devices are generating massive volumes of user data. Processing this data can reveal valuable insights. For example, a wearable device collecting its user's vitals can use the collected data to provide health advice. Typically the collected data is sent to some remote computing resources for processing. However, due to the vastly increasing volumes of such data, it becomes infeasible to efficiently transfer it over the network. Edge computing is an emerging system architecture that employs nearby devices for processing and can be used to alleviate the aforementioned data transfer problem. However, it remains hard to design and develop edge computing applications, making it a task reserved for expert developers. This dissertation is concerned with democratizing the development of edge applications, so the task would become accessible for regular developers. The overriding idea is to make the design and implementation of edge applications more systematic by means of programming support, exemplary use cases, and methodologies.

Dedication

To my family, for their unwavering love and support.

Acknowledgments

My dissertation would not have been possible without the support of many people. I would like to express my deepest appreciation to all who have helped me completing my Ph.D.[1]

First, I would like to express my gratitude to my advisor, Dr. Eli Tilevich, for his continued guidance, support, and encouragement. He gave me the freedom to pursue my research ideas and was always available for brainstorming new ideas. He has been a mentor, a worthy table tennis rival, and a friend.

I would also like to thank my committee members, Drs. Scott McCrickard, Na Meng, Rafael Montalvo, and Francisco Servant, whose constructive feedback and invaluable insights helped me improve the overall quality of my dissertation.

I very much appreciate the constructive criticism, invaluable insights, and table tennis lessons that my lab mates, Jason, Karn, Kijin, and Yin, gave me.

Thanks also to Lindah Kotut and Timothy Stelter for their invaluable feedback and support. I also very much appreciate the support that Sharon Kinder-Potter, Robert Marcum, and Teresa Hall provided throughout my studies at VT.

Thanks also go to Russell Fritzemeier, Brandon Semel, Joshua Moser, and Gregor Kildow for valuable advice and practical suggestions. Arnab - thank you so much for believing in my ideas and for the hard work you put into making them a reality. I hope we can have many more future collaborations.

I would like to acknowledge the help of all my mentors at Cisco Systems, who provided support, practical suggestions, and helpful advice: Dr. Montalvo, Chris Olson, Patrick Nau-

[1]This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1717065, "SHF:CSR:Small: Perpetuum Mobile: Orchestrating the Provisioning of Pervasive Resources for Emerging Mobile Applications", a Cisco Research Award, and Capes foundation Science Without Borders program, process number 13003-13-5. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of NSF, Cisco, or Capes.

rayan, Kevin Beeman, and Jim Warren.

Last, but not least, I am deeply indebted to my parents, Eliete and Rogério, who are the strongest people I know, and I pray that God can reward you for all the support you've given me. My sister Sara, her constructive criticism and practical suggestions. I am also grateful to aunts Elizete, Elizangella, and Eliã for their unwavering support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern mobile, IoT, and wearable devices have been collecting massive amounts of sensor data that must be processed efficiently. Relying on cloud resources to process all the collected data is no longer viable, due to the network bandwidth bottlenecks and privacy concerns. Edge computing offers a solution to this problem by processing this data locally at the edge of the network by means of nearby devices. By doing so, edge applications offer several advantages, such as removing network transmission bottlenecks and alleviating certain privacy concerns [20, 36, 40]. For example, a health monitoring app can pre-process a user's vitals locally at the edge, before uploading the results to a cloud server for storage, thus reducing the required network load and keeping the user's raw data on the device. Examples of successful applications of edge computing include deployments of smart, IoT, and augmented reality systems [20, 65, 82].

Because edge computing differs from the established programming conventions and paradigms of cloud computing, developers find it hard to build and maintain edge applications [5]. What stands in the way of developers is the steep learning curve required to master the challenges of device-to-device communication and addressing the resource scarcity of edge-based devices [28, 79]. To meet these challenges, developers often resort to ad-hoc software development practices, introducing low-level, brittle, and hard-to-maintain code to the codebase. This code is error-prone, expensive to maintain, and vulnerable in terms of security. Also, it remains hard to estimate the performance and energy consumption of edge devices for a given

application [16]. The vision of this dissertation is that rather than relying on ad-hoc implementation practices and guesswork, developers of edge applications should be equipped with powerful middleware mechanisms and performance estimation tools that would allow them to efficiently utilize the available edge resources [17, 18]. The findings of this dissertation research make inroads into putting this vision into reality.

1.1 Major Research Contributions and Scope

The thesis of this research is that modular middleware abstractions, exemplar use cases, and ML-based performance estimation can make the design and development of edge applications more systematic. Hence, our goal is to provide: 1) middleware to support remote-component data access while ensuring remote-data encapsulation; 2) exemplary use cases of edge-based ML applications; 3) performance estimation methodologies to guide developers on resource utilization of their ML edge-based applications. To achieve these goals, this dissertation contributes three key interrelated research thrusts:

1. **To innovate in the middleware space in order to reduce the complexity of implementing device-to-device communication and coordination logic.** Software and systems research in the middleware domain traditionally has focused on abstracting way the presence of distribution altogether [98], as exemplified by ubiquitous Remote Procedure Call (RPC). However, our position is that these contextual differences should be made evident to developers, when it comes to transferring data across the network. To that end, we design novel middleware that ensures encapsulation and makes evident, to the distributed application developers, how the data transfer patterns would impact the runtime performance [17, 19].

2. **To investigate a real-world use case of edge-based ML tasks.** Participatory sensing leverages users' smartphones to collect and share sensor data from their surroundings [9, 62] to effectively capture environmental and behavioral information for analyzing data analysis and discovering knowledge [45]. Traditionally, participatory sensing architectures transfer collected data to the cloud for processing. However, our position is that edge-based processing can be used to reduce network bottlenecks and improve data privacy for participatory sensing. The findings of this research put forward a novel architecture for participatory sensing that fully leverages edge-based resources, thereby removing the network transmission bottleneck [20].

3. **To inform edge-application developers about the expected performance and resource utilization of their completed solutions in the wild.** The performance and resource utilization of edge applications are hard to estimate. Therefore, developers cannot easily estimate how their edge-deployed code would perform under different workloads, a particularly important issue due to the resource scarcity of edge environments [85]. Specifically, to estimate the performance of edge-based clustering, developers need to understand not only the intrinsic properties of clustering algorithms, but also the expected resource utilization trade-offs. Also, because of development budget limitations and non-trivial repeatability due to a lack of controlled environments, developers cannot always access real test beds [35]. However, our position is that these factors should not prevent system developers from implementing their intended solutions with confidence. Deep-Learning can be used to accurately estimate the performance and resource utilization of edge-based applications. To demonstrate the feasibility of this methodology, we benchmarked four variants of an edge-based clustering application and used the information to train a DL model for predicting the performance and resource utilization of edge-based clustering applications [16].

1.2 Broader Impact

We are on the verge of the next technological leap. All parts of modern society are generating massive volumes of data, which contain valuable information mined for actionable insights that can improve our lives. For example, regular users are generating massive amounts of media files that can be shared with large audiences via social networks. With IoT devices, factories monitor the production process and use the collected data for optimization and planning. However, relying solely on cloud-based resources to process all these massive amounts of data causes serious network issues (e.g., network bottlenecks and service interruptions). Edge computing can solve these issues by processing much of that data locally on nearby devices. Therefore, edge computing may offer solutions to problems that currently prevent new technologies from taking off (e.g., autonomous vehicles, drone swarms, smart buildings, etc.). However, the development of edge applications is extremely hard, and, as such, it remains the domain of highly experienced system developers.

The goal of this dissertation is to democratize the development of edge applications, so it would become accessible for regular developers. To that end, this dissertation puts forward programming support, exemplary use cases, and methodologies, intended to make the design and implementation of edge applications more systematic. By bringing proven software engineering solutions to this emerging domain, we hope to empower developers to create advanced edge applications required to make the next technological leap.

1.3 Structure

The rest of this dissertation is structured as follows. Chapter 2 provides the background and related work. Chapter 3, 4, and 5 cover our programming support for inter-device

communications, exemplary use case on using participatory sensing to reduce the network requirements of deploying edge-based ML tasks, and performance estimation of edge-based ML tasks. Chapter 6 presents a discussion on the applicability of the contributions of this work. Chapter 7 presents the concluding remarks, summarizes the contributions of this research, and discusses future work directions. Appendix A.1 presents additional benchmark measurements from our work on performance estimation. Appendix B presents possible direction for user studies that explore usability aspects of our contributions. Appendix C presents step-by-step guides on how run the code used in Chapter 5.

Chapter 2

Background & Related Work

In this chapter, we introduce the technical background required to understand our conceptual contributions. Also, we provide the related work to our research contributions.

2.1 Background

In this section, we introduce the technical background required to understand our conceptual contributions. The reader knowledgeable with key Android APIs, distributed programming concepts, Deep-Learning, clustering, and participatory sensing can safely skip this section.

Android Components: Each installed application in Android (i.e., *.apk file) typically runs in a separate process and can be composed of *Activities*, *Services*, *Content Providers*, and *Broadcast Receivers*. These four components communicate through messages called *Intents*, which are routed by the Android runtime. *Activities* are graphical components that provide a user interface to the client. The main *Activity* is instantiated upon the user launching an application. *Services* run in the background to execute long-running tasks. *Content Providers* manage access to persistent data. *Broadcast Receivers* receive and react to event notifications.

Inter-Component Communication (ICC): Android applications are structured as a

collection of components, each of which represents a reusable unit of functionality. An application can also use components of other applications, both within the same process or across processes, as long as the permission scheme in place allows it. Loosely coupled components interact with each other via ICC messages, represented as reusable `Intent`[1] objects [58, 73]. When interacting via ICC, Android components can operate both within the same application and across different ones. To render a component accessible to other applications, a developer sets its `exported` attribute to `true` in the manifest file[2].

Remote Method Invocation: Remote Method Invocation (RMI) provides remote procedure call (RPC [71]) functionality to Java applications. It enables any Java object on one JVM to invoke methods of an object running on another JVM [101]. The RMI abstractions render local and remote invocations almost identical to each other. The only difference is that remote methods must be declared as throwing `RemoteException`, thus requiring that the developer provide logic for handling partial failures, which are omnipresent in distributed computing.

The Remote Procedure Call (RPC): RPC provides programming abstractions and runtime support to make invoking methods in a different address space as similar as possible to invoking those in the same address space [7]. In distributed computing, RPC transfers each remote call from a client to execute on a server, while returning back the results. Examples of object-oriented variants of RPC include DCOM [8], CORBA [97], and Java RMI [43]. Mobile Plus [74] provides an RPC-based middleware to enable Android devices to share resources; it does so by modifying the Android OS, so as to properly support all the required functionalities with adequate performance. Since one of the key design objectives of RICCi is portability, our reference implementation runs entirely in user space and requires no changes to the Android platform.

[1] https://developer.android.com/reference/android/content/Intent.html
[2] http://developer.android.com/guide/topics/manifest/manifest-intro.html

In Android, each process runs in a separate address space. To enable processes to communicate with each other within the same device, Android features the Binder framework, which provides RPC-based communication between client and server processes. Binder passes data to remote method calls, returning the results to the client's calling thread. Android even offers Android Interface Definition Language (AIDL)[3], similar to CORBA IDL, for developers to define a set of remote methods to be invoked between client and server processes [30, 31, 89].

Despite the ubiquity of RPC, this distributed programming paradigm has been critiqued as not realistically reflecting the actual differences between the local and remote computing models [98]. In a way, the RPC abstraction may be too powerful, making remote calls look indistinguishable from local calls, and thus encouraging the questionable "papering over the network" designs for distributed applications [47]). Because remote calls can take orders of magnitude longer than local calls to execute, treating them uniformly is only possible in low-latency distributed systems with high resource availability. Perhaps, that is why RPC-based systems have been particularly successful as a communication mechanism between different processes on the same machine.

Participatory sensing: is a methodology for collecting environmental and behavioral data by leveraging personal mobile devices [69]. Participatory sensing avoids the necessity of deploying and maintaining sensors, while the ubiquity of mobile users and their diverse mobility patterns enable collecting data from a variety of environments [45]. However, participatory sensing tends to generate large multimedia files or large collections of small-size sensor readings, types of data that are known to overwhelm the available network bandwidth, clogging the transmission [81].

Deep Learning and Performance Estimation: Deep Learning [56, 88] (DL) is an

[3]https://developer.android.com/guide/components/aidl.html

ML methodology that uses artificial neural networks (ANN), inspired by the structure of neurons in the human brain. DNNs have become indispensable tools in fields including image and video recognition, natural language processing, etc. DNNs combine the advantages of DL and neural networks and can solve nonlinear problems more satisfactorily compared to conventional ML algorithms. Prior work has applied ML techniques to estimate the performance of edge devices [60] and applications [33, 42]. DL has been applied widely to solve problems in multiple domains [4, 15, 21]. DL methods can deliver performance and accuracy greater than those of ML ones [52, 72]. Stargazer's prediction heuristic is driven by DL to estimate the performance of devices executing clustering algorithms in the edge.

Clustering: Clustering refers to techniques that group a set of items with similar attributes [53]. Clustering has been the subject of numerous research efforts: a Google Scholar search for the word "Clustering" returns over 770,000 publications only since 2015. With its primary application to exploratory data mining, clustering is used in domains including pattern recognition, image analysis, and information retrieval [38].

2.2 Related Work

In this section, we provide the related work of our research contributions. Also, it would be unrealistic to compare and contrast our work with all these prior efforts. Hence, we limit the related work coverage to the most relevant or recent research works.

2.2.1 Programming Support

We have designed RICCi to support a distributed programming model that is distinct from RPC. Instead, RICCi extends the Inter Component Communication (ICC) framework,

which is fundamentally asynchronous and event-based. When implementing RICCi-based applications, developers are required to take into account the remote data' properties (i.e. its size and privacy) in order to determine how and whether to transfer it across the network. As a result, RICCi is not an RPC-based but a remote data-based distributed programming abstraction. Internally, RICCi does make use of RPC to implement its remote access data exchange pattern, in which the data remains stationary and instead accessed by means of remote callbacks, implemented using an extant RPC system.

Regarding related work on middleware approaches for resource sharing, ShAir [22] is a latency-tolerant data sharing approach that uses P2P communication for distribute mobile applications. While similar in its objectives, RICCi maintains strong encapsulation for P2P communication, in which the shared resources remain in the form of components. RICCi also shares goals with Sip2Share [10], which is an Android framework that extends parts of the built-in mechanism for sharing services and activities. A distinguishing characteristic of RICCi is the precise control over how remote data is transferred across the network; it provides this control by means of three declarative data exchange patterns, which specify how exactly to transfer remote data. μMAIS [78] also focuses on similar objectives, but unlike RICCi, it exports distributed resources in the form of Web Services. Mist [92] follows a publish-subscribe paradigm to provide a latency-tolerant middleware framework. Similarly to RICCi, Mist focuses on delivering *data elements* across the network. However, RICCi also differentiates its data delivery policies based on the type and volume of the transferred data. MobiClique [76] leverages nearby existing social network contacts to form ad hoc networks for data sharing. Following an ad hoc approach to form networks or to share component data can be an interesting future work direction.

Another popular technique used for optimizing mobile applications is code offloading, which relies on the opportunistic execution of code by remote servers [27, 87] or on au-

tomated application partitioning [94]. Multiple offloading strategies have been proposed to enhance smartphone applications with cloud-based resources [11, 14, 26, 51, 54, 55]. Massari *et al.* [67] propose to use code-offloading techniques to use available computational power from discarded cellphones. RICCi can serve as a middleware platform for offloading local components to run on a remote device, an idea we plan to explore as a future work direction.

In terms of network-aware applications, Pinte *et al.* [77] study several major techniques and present examples of adding network-awareness and distribution to mobile applications. To explore how to access distributed Android resources, Nakao *et al.* [70] study how feasible it would be to use the inter-process communication mechanism to invoke remote services.

2.2.2 Exemplary Use Case: Participatory Sensing

There are research works that have explored how to leverage edge computing to process raw sensor data. These works use devices near where the data is being generated to alleviate network utilization [37, 66, 102]. However, these approaches require deploying additional edge devices [66].

Our contribution provides an architecture for a variety of participatory sensing tasks. Although we evaluate our approach in its ability to execute one specific scenario, our architecture can support any sensing task, as long as it can be expressed as a collection of microservices. Also, our architecture provides the necessary administrative procedures for participatory sensing, such as reward distribution and data validation [29, 83], by means of edge-based resources.

2.2.3 Performance Estimation

The performance of edge applications has been estimated using simulators [35, 44] as well as static and dynamic program analyses [1, 2, 12]. Our approach differs by not relying on simulation but instead estimating performance from the source code via a DNN. In addition, Stargazer estimates not only energy consumption but also additional performance metrics (e.g., % CPU usage, % memory utilization, etc.). To ascertain the complexity of ML tasks under evaluation, Stargazer uses Singularity in a black-box fashion [99]. To find the correlations in the collected data sets to train Stargazer, this work uses linear and polynomial regression models. A linear regression model applies linear predictor functions to estimate model output parameters from the data [48]. A polynomial model regression fits a nonlinear relationship between the data and corresponding output parameters via an nth degree polynomial [23].

Despite a wide variety of clustering algorithms, our exploration focuses on the ones most widely used and offering dissimilar performance characteristics. Specifically, we study *K-Means*, an unsupervised learning technique that assigns n elements to k clusters, thus allocating each element to the cluster with the nearest mean value [64]; *Farthest First Clustering*, a variant of K-Means, based on heuristics, that chooses centroids and assigns the elements in the cluster at the farthest point from the existing cluster center within the data area [6]; *Expectation-Maximization* (EM), a probability-based technique, that assigns elements to clusters based on a probability model rather than a distance metric, like in K-Means. EM partially assigns elements to different clusters [68]; *Density-Based Spatial Clustering of Application with Noise*, a density-based technique, that identifies clusters by analyzing the concentration (higher point density) of data points, often resulting in arbitrarily shaped clusters, as compared to K-Means [24]. This work is not concerned with distributing, scaling, or optimizing clustering algorithms. Our goal is to understand the

performance characteristics of existing clustering algorithms deployed in the edge.

Chapter 3

Programming Support

The Android platform is inherently component-based, with mobile apps comprising sets of interacting components. Each application component encapsulates a distinct functionality that can be accessed by external clients via a simple interface. However, this component-based encapsulation breaks when devices need to share data. Implementing data-sharing functionality requires application developers to write low-level network communication code that directly manipulates the data. The resulting communication code is not reusable and hard-to-maintain.

Mobile, IoT, and wearable devices generate massive amounts of data, which needs to be transferred to other mobile devices. For example, a health monitoring app, running on a wearable, periodically reads its wearer's vital signs, which are then displayed on an app, located on the wearer's smartphone or tablet. The vitals can be transmitted either directly, peer-to-peer, or via a cloud-based server. However, the resulting information flow is device-to-device, a common pattern for a growing number of distributed mobile applications.

In this chapter, we put forward a middleware framework that preserves the encapsulation of the Android component model when sharing data across devices. We evolve the ubiquitous Android Inter-Component Communication (ICC) into Remote ICC (RICCi), with the RICCi API mirroring that of ICC and also providing a declarative interface for specifying how to transfer component data (i.e., `Intent` objects) across the network. RICCi enables mobile application developers to use components, hosted on remote devices, through an API that

resembles as close as possible that for using components hosted on the same device. Unlike RPC-based middleware platforms (e.g., CORBA [97], gRPC [32], Java RMI [43], etc.), the RICCi programming model is not procedure-oriented but is data-centric. That is, rather than modeling distributed communication as a procedure call, RICCi introduces the concept of *remote component data*, which is transferred across devices following a given data exchange pattern. In that way, RICCi bears similarity to the RESTful architecture [25], albeit with strong encapsulation and fine-grained control over how the data is transferred from the source to the destination.

RICCi requires developers to treat latency as a first-class design concept. The RICCi API includes a declarative interface for specifying how and even whether remote-component data should be transferred across the network; a component's data can be *copied*, *streamed*, or remain *remote*. Therefore, developers cannot disregard the differences in latency between the communication within the same device and those across different devices over the network. Thus, RICCi makes it impossible for developers to "paper over the network"[47].

In addition to reusability and encapsulation, the reference implementation of RICCi provides all the necessary low-level networking support and can route the distributed communication either directly, peer-to-peer, or via a broker. The broker role can be played by either an edge server, connected to devices via a local area network, or a cloud service, connected through a WAN. Also, RICCi is fully portable, as it works with any standard Android installation, without requiring any changes to the Android SDK.

For the evaluation, we implemented two versions of distributed benchmark applications. For the first, we used our reference implementation of RICCi, and for the second, we implemented using the existing RPC middleware platform. Based on our evaluation, RICCi is comparable in terms of the resulting performance (i.e., energy consumption) while requiring fewer lines of code and lower code complexity. Despite our reference implementation being

Android-specific, the insights gained from this work can be applied to other mobile platforms, enabling distributed devices to efficiently interact with each other at the component level.

The contributions of this chapter are as follows:

1. We describe the design and implementation of RICCi — a novel middleware framework for intuitive and efficient sharing of component data across Android devices; unlike existing RPC-based middleware platforms, RICCi is structured around the concept of *remote component data*.

2. We empirically evaluate the RICCi programming model in terms of performance and expressiveness.

3. We present guidelines for application developers on how to use RICCi effectively in the development of emerging distributed mobile applications.

In Section 3.1, we present three motivating use cases for our solution; in Section 3.2, we present the design of the RICCi programming interface, its architecture, middleware, and broker systems. In Section 3.3, we explain the implementation insights of our solution. In Section 3.4, we explain how RICCi streamlines the implementation of the motivation scenarios. In Section 3.5, we evaluate the performance and software engineering characteristics of RICCi. Finally, in Section 3.6, we outline future work directions and present concluding remarks.

Background and related work is available in Chapter 2. Chapter 6.1 presents the applicability and limitations of our solution. Also, the Appendix B.1 contains possible usability study directions that practitioners may follow.

3.1 Distributed Data Sharing Scenarios

In this section, we present three examples of mobile distributed applications to demonstrate the need for a middleware platform that can streamline their implementation.

3.1.1 Sharing Documents Across Mobile Devices

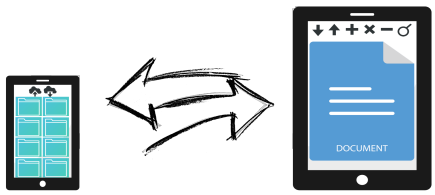

Figure 3.1: Copying files from smartphone to tablet

Mobile applications are often used in environments with limited network connectivity. Business professionals on the move often find themselves in locations, in which they cannot connect to cloud-based services for availability, bandwidth, or security reasons. Users can take notes on one device but edit them on another device that runs a proprietary document editing application (e.g., MS Word). Consider having to develop a document management application that synchronizes documents across multiple mobile devices (Figure 3.1). In the presence of a reliable network connection, a cloud-based solution can satisfy the requirements. However, in the absence of the standard network infrastructure, devices should be able to communicate with each other directly or possibly brokered by a LAN-connected edge server. Mobile application developers would like to be able to design and implement such applications without having to write special-purpose logic that addresses the aforementioned dissimilar network conditions. In addition, since Android uses components to access disk files, developers are likely to find it advantageous to retain the component-based representation for the documents when transferring them across the network.

3.1.2 Reading Input from IoT Devices

Private residences routinely feature several IoT devices installed in different locations. These sensors monitor temperature, noise, humidity, as well as provide video feeds and check for package deliveries. A house resident can use a mobile device to receive periodic readings from these IoT devices.

Figure 3.2: Streaming video data from IoT cameras to phone

If the Internet bandwidth is plentiful and readily available, a cloud-based service can be used to mediate the interaction (Figure 3.2). However, if the bandwidth is limited or expensive, developers should be able to express that the application is to use an edge server, connected via a local network, or that the devices are to communicate with each other directly peer-to-peer. Despite the complexity of having to support multiple network environments, developers may still prefer to follow a component-based design for such applications, with both the media data and its presentation logic represented as separate components. In other words, the application's business logic should not be polluted with low-level system functionality that selects the appropriate communication mechanism based on the network conditions in place. That is, the logic required to select the most suitable distributed communication pattern should be separate from the application's business logic.

Figure 3.3: Gauging the production process by remotely accessing properties of immovable data

3.1.3 Reading Properties of Temporal Data

In complex industrial environments, mobile devices are often used to monitor different complex processes and aggregate readings from various sensors. Due to the data accumulating rapidly and being of relevance only temporarily, it may be disadvantageous to transfer this temporal data to remote servers for storage, thus deeming the data unmovable. However, various interested parties within the company may still want to query the accumulated data for some of its real-time properties, and they want to do so remotely from their mobile devices. For example, an employee can be carrying a tablet that receives input from the sensors installed throughout a manufacturing floor (Figure 3.3). A factory manager may want to inquire about some real-time aspect of the production process, while at a meeting in a different office. Notice, that the data is intended to be available to multiple users, as the mobile device collecting sensor data is *not* for personal use. The employee carrying the device around the manufacturing floor assumes that the collected sensor data from the floor will be accessible to the management at any time. To that end, consider having to develop an app capable of querying the unmovable data on the employee's tablet without having to transfer the data itself, with the manager's mobile device providing a viewing interface. Furthermore, developers would prefer to follow a component-based design, with the immovable data and presentation logic implemented as separate components, communicating with each other over the local network.

3.2 Design Overview

This section provides an overview of RICCi and discusses how it enhances ICC for inter-device component data sharing. Also, this section explains how RICCi works at runtime.

Our design requirements are two-fold. First, we aim at providing an inter-device programming abstraction that can be quickly mastered and put into practice by developers familiar with the Android abstractions for component interaction on the same device. Second, we want to ensure that the new abstraction sufficiently accommodates for the differences between the *same-device* and *device-to-device* computing environments, with a particular emphasis on recognizing the presence of latency, which is known to increase by orders of magnitude when switching from local to distributed computing environments. This recognition is essential to ensure that the resulting distributed mobile application maintains the required quality of service.

By designing RICCi, we want to enable Android developers to use the abstraction to build efficient distributed applications, while reasoning at the component level. Without properly designed programming abstractions, typical components sharing scenarios, such as the aforementioned, would require developers to write low-level code for each piece of the distributed functionality (e.g., network data exchange patterns, data marshaling, handling faults, etc.). Even though the actual exchanged data is naturally represented as Android components, the attendant encapsulation benefits would quickly disappear if these components' data is transferred across the network without proper abstractions. In contrast, RICCi enables developers to continue reasoning at the component level when implementing inter-device interactions.

3.2.1 General Design

RICCi evolves the built-in Android ICC programming paradigm, whose overriding design principle is to enable developers to integrate ready-made components into their applications, thereby reducing the programming effort and streamlining the development process. Hence, our design objective is to retain the software engineering benefits afforded by ICC, but also to extend them to environments that comprise multiple Android devices. In other words, we would like to enable Android developers to leverage their familiarity with ICC to easily implement distributed applications capable of accessing components hosted by other Android devices.

```
1 public void getDocument() {
2       Intent intent = new Intent(
3       Action.OPEN_DOCUMENT);
4       intent.addCategory(CATEGORY_OPENABLE);
5       intent.setType("text/plain");
6       startActivityForResult(intent, code);
7 }
```

Figure 3.4: Accessing document information with ICC

```
1 public void getDocument() {
2       RemoteIntent intent = new RemoteIntent(
3       Action.OPEN_DOCUMENT, Transfer.COPY);
4       intent.addCategory(CATEGORY_OPENABLE);
5       intent.setType("text/plain");
6       startActivity(intent);
7 }
```

Figure 3.5: Accessing document information using RICCi

Figures 3.4 and 3.5 show two code snippets, in which an application requests a document stored by an Android device. Figure 3.4 shows the code that uses the built-in ICC to request a document stored on the same device. Meanwhile, Figure 3.5 shows an equivalent

RICCi implementation that requests a document stored on another device. As one can observe, these code snippets are remarkably similar. In fact, the only difference is that the RICCi version uses type-compatible `RemoteIntent` objects (line 3 of Figure 3.5) instead of `Intent` objects (line 3 of Figure 3.4). The `RemoteIntent`'s constructor take an additional parameter, `Transfer.COPY`, thus declaratively specifying the *by-copy* semantics for the document component's data. We have elided the required handling of remote exceptions that the distributed runtime would raise in response to detecting various failures.

Of course, the original ICC mechanism cannot be reused as is in a distributed setting. As mentioned above, distributed applications typically experience higher latencies than centralized applications. To accommodate the increased latency, our design requires that the developer specify how the component data is to be transferred across the network. Specifically, depending on the properties of component data, it can be either copied, streamed, or retained in place to be accessed remotely. `Intent` objects, encapsulating small data volumes, can be copied across the network efficiently; `Intent`s representing media files can be streamed to be played on remote devices; while `Intent`s with platform-specific dependencies are "anchored" to their devices, accessed by means of remote callbacks from the accessing devices. The RICCi API includes remote exceptions that reflect various partial failure conditions, likely to occur in different deployment environments. The developer using RICCi will have to handle these failures in an application-specific manner.

Naturally, when information is transferred across the network, the issue of *security* comes to the fray, and the developer must make appropriate provisions. Because RICCi enables component-level distributed programming, it promotes the encapsulation principle. Encapsulated functionality is easier to systematically enhance with additional properties than scattered functionality. In the RICCi architecture, it is known exactly when distributed interactions commence and end, and in which patterns the data is exchanged. This foreknowledge

simplifies the provisioning of various security enhancements, such as adding encryption and following secure network transfer protocols.

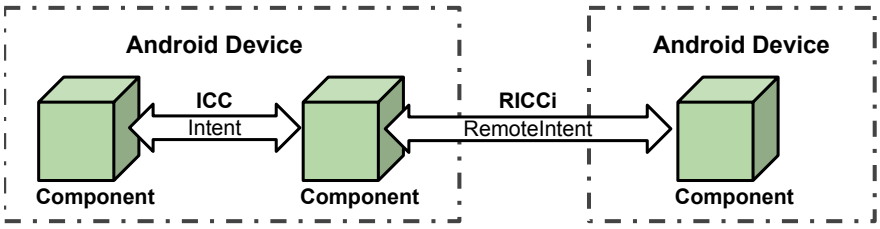

Figure 3.6: ICC vs. RICCi

Figure 3.6 shows a system diagram demonstrating the differences between ICC and RICCi.

3.2.2 System Architecture

Figure 3.7 shows an overview of a distributed mobile application developed using RICCi. In this application, the *Source Device* is the device that is requesting to access a component on another device. To that end, the *Source Device* declares and sends a `RemoteIntent` object. The *Target Device* is the device that receives and handles the request to access the component in question. The *Broker Server* is responsible for handling the connection between *Source Device* and *Target Device*. Depending on the *Broker Server's* location (LAN or WAN), the broker is also responsible for relaying all the transmission between the connected *Source* and *Target* devices. We provide additional details about the Broker Server below. When devices are to communicate peer-to-peer, RICCi assumes that the devices have been properly configured for such interactions. Under these scenarios, the RICCi runtime handles all requests and responses without the aid of a broker.

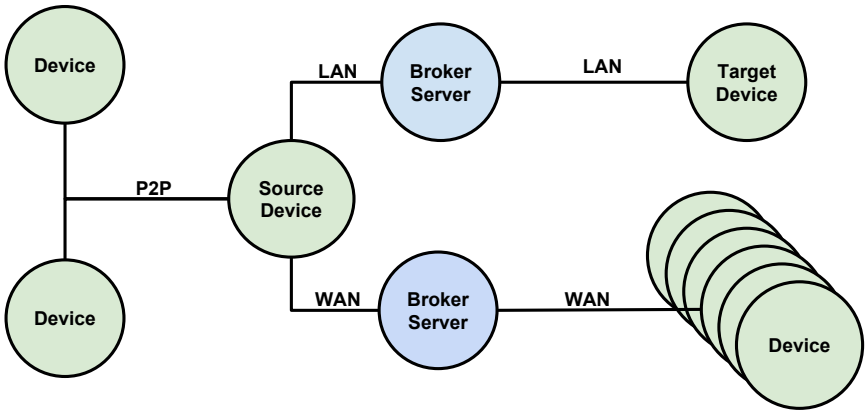

Figure 3.7: RICCi Architecture

RICCi Dataflow

When sharing component data across devices using RICCi, the process comprises three main phases: 1) The *Source* device creates a `RemoteIntent` (with the desired target information and data exchange patterns) and sends it to the *Target* device through a network. 2) The *Target* device receives the `RemoteIntent`, executes its defined action, and returns the results, following the specified data exchange pattern to the *Source* device. 3) The *Source* device receives the results in a fashion prescribed by the specified exchange pattern.

3.2.3 System Architecture Components

RICCi runtime includes both the functionality executing on each of the participating devices as well as the distributed logic required for interacting with broker servers.

Device components:

RICCi runtime is responsible for processing the component data sharing requests within a device. The runtime is also responsible for setting up a connection to Broker servers,

which route the requests between the devices. We provide additional details on how RICCi connects to broker servers in Section 3.3. RICCi activities in both *Source* and *Target* devices comprise two threads responsible for managing incoming requests: a Messenger Thread and a Handler Thread. The Messenger Thread interacts with a transmission channel by sending and receiving data. The incoming data is transmitted to the Handler Thread. The Handler Thread is responsible for processing incoming requests and generating the appropriate responses, which are then retransmitted to the intended device by the Messenger Thread.

The Broker Server:

The Broker server is responsible for delivering component data to the intended destination. Broker servers are divided into two categories: edge (LAN) and WAN brokers. A broker server is responsible for keeping track of the connected devices, resources, and component data that these devices have available for sharing. A broker server maintains a list of devices currently connected, and it analyzes the transferred messages, specifically the `Bundle` objects of the `RemoteIntent`, to determine their destination and data exchange patterns. RICCi runtime uses a priority queue, explained in Section 3.3, to determine to which broker a given device should connect at any given time. Broker servers connected via local networks are responsible for pairing up devices and setting a peer-to-peer connection between the paired devices. Each pair of devices has a Requester (making the requests) and a Provider (providing the resources). In terms of behavior, LAN brokers are slightly different from WAN brokers. LAN brokers are only responsible for pairing up the communicating devices. RICCi establishes direct connections for streaming and accessing data remotely. WAN brokers also pair up the communicating devices and are responsible for acting as a gateway, relaying all the data transferred between any two interacting devices.

3.3 Implementation

In this section, we present additional technical details of the reference implementation of RICCi.

3.3.1 Data Exchange Logic

In abstract terms, RICCi facilitates the development of distributed Android applications. To that end, RICCi manages the connection between the interacting devices and the specifics of how the component data is transferred across the network. RICCi runtime transfers, serializes, and compresses component data, while handling the connection setup between the devices.

`RemoteIntent` objects are used by the reference implementation to transport component data in a way prescribed by a given data exchange pattern. `RemoteIntent` and its contents are serialized using GSON [1] to be represented as JSON objects. When created, a `RemoteIntent` can receive a *data exchange pattern* and an *action*. Actions define what kind of operations the `RemoteIntent` will perform, while data exchange patterns define how the results will be transmitted. Figure 3.8 shows the declaration of a `RemoteIntent` with an action (`Intent.ACTION_PICK`) line 2 and a data exchange pattern `Transfer.Stream` line 3. Developers can choose between three data exchange patterns: `COPY`, `REMOTE` or `STREAM`. The created `RemoteIntent` can be transmitted to a Target device by calling the RICCi implementation of the `startActivity()` method, which processes and sends `RemoteIntents` across the network by means of the sockets API.

The *Target* device (i.e., providing the resources) receives the `RemoteIntent` and tags it as an incoming message and forwards it to an `IntentService`. Developers are required to

[1] https://github.com/google/gson

```
1  RemoteIntent rIntent = new RemoteIntent(
2              Intent.ACTION_PICK, Transfer.STREAM);
3  startActivity(rIntent);
```

Figure 3.8: RemoteIntent declaration

add the `Intent- Service` extending `RICCiIntentService` as a service in the manifest file, so the Android system would allow RICCi runtime to run. This service is responsible for determining how to handle `RemoteIntent`s according to their data exchange patterns. After determining how to transfer the requested component data, the `IntentService` sends the contents of the `RemoteIntent` to a modified `BroadcastReceiver`, which calls the method `startActivity ForResult()` with the contents of the `RemoteIntent`, and then returns standard return codes. The results are returned to the `IntentService`, which follows the specified data exchange pattern to transmit the results back to the Target Device.

3.3.2 Data Exchange Patterns

When specifying the **Copy** data exchange pattern, the developer creates a `RemoteIntent` with parameter `Transfer.`COPY. This parameter is used by the RICCi runtime and by the Broker server to handle the requests. The Copy data exchange pattern signals RICCi runtime to return an exact copy of the data from the resulting operation of a component in the target device. The contents of the resulting `Intent` are placed inside a `Bundle` object in a `RemoteIntent`, which is sent back to the `Source` device. For example, RICCi runtime collects the *data* `Intent` resulting from the `onActivityResult`, which references context-specific information stored at the target device. RICCi then uses the reference stored at the *data* `Intent` to directly access all the information that describes the user contact, such as number, email, etc., and it sends a copy to the `Source` device.

When specifying the **Stream** data exchange pattern, the developer creates a `RemoteIntent` with parameter `Transfer.STREAM`. This parameter is used by both the RICCi runtime and Broker server as metadata, to determine how data should be accessed. In the case of a WAN broker, this metadata also signifies that the server needs to gateway the incoming data stream. The Stream data exchange pattern signals the *Target*'s RICCi runtime that it needs to access a file and stream its contents. For example, if the resulting `Intent` is a reference to a media file, the `Target`'s RICCi runtime locates the file, creates a `ServerSocket`, and sends its IP and port to the `Source` device. The `Source` device then connects to the server and requests the file transmission. Finally, the `Target` device serializes the accessed media file and starts the process of transmitting it to the `Source` device.

When specifying the **Remote** data exchange pattern, the developer creates a `RemoteIntent` with parameter `Transfer.REMOTE` and a `RemoteAssistant` object. The `RemoteAssistant` is used by RICCi runtime to execute the defined remote operations, while the `RemoteIntent` identifies the *Target* device's component. The `Target` device specifies a port number and waits for the `Source` device to interact with the object. RICCi runtime provides a gateway to perform remote operations.

For the implementation of this feature, we used LipeRMI [3] as the current version of Java RMI is not supported by Android OS. LipeRMI is a plug-in replacement for the native Java RMI facility for the Android platform. The `RemoteAssistant` object is used by the *Source* device's RICCi runtime to perform the remote method invocations. The results of those operations are placed in a `RemoteResultsHolder` object, which is made available for access after the interaction is completed. By default the contents of the transmission are placed in a temporary file for streaming, however, the developers can modify RICCi's implementation to save the file in the device.

We are aware that developers may modify and enhance their applications to provide greater

functionality and flexibility for usage of the remote pattern. To that end, RICCi features a code generator that synthesizes code compatible with the RICCi remote data exchange pattern, including both classes and the manifest file. The code generator enables developers to explore the features of RICCi to a greater extent to provide the desired method invocations. The code generator receives as input an `ImmovableIntent` object and generates a stub and skeleton classes that serve as client and server-side proxies, respectively. Their methods provide the logic required to dispatch a remote callback method on the Source Device and dispatch them to the Target Device. The generator needs to be rerun every time the number or type of methods in a given `ImmovableIntent` class changes.

3.3.3 Network Topology Selection

One of the features of the RICCi runtime is to select the most advantageous network topology for two Android devices to exchange component data. Recall that the devices can interact either directly peer-to-peer or via a broker, provided by either a LAN-connected edge server or a WAN-connected cloud-based service. To determine which of the three topologies to select for a given inter-device component sharing, RICCi follows a predetermined set of customizable priorities. However, developers can change and customize the network selection logic at will. To that end, RICCi provides class `RicciAppCompatActivity` that developers can extend, providing a custom implementation of method `connectWebSocket ()`. This method receives an array of IP addresses of the available servers as input, from which it initially connects its device either directly to another device or to a broker server. In the reference implementation of the method, `connectWebSocket()` sequentially checks over each provided IP for the first server available to serve as a broker. Developers using RICCi are required to call the `connectWebSocket()` within the method `onCreate()` in the `MainActivity` class of their distributed mobile application.

The RICCi runtime sets up the connection topology between a device and a broker as follows. First the device calls the `connect-WebSocket()` method to check the list of possible available servers (1 & 2). If the server is available, RICCi performs a handshake operation and connects to the broker server (3), in which it provides its current device information, IP address, and possible capabilities to the server. The server receives the setup requests and persists the device's information (4). Finally, the server concludes the setup process and starts awaiting requests (5). In case of a request, the server goes over the list of connected devices and creates a pair **requester** (device making a request) and **provider** (device providing the information). Based on the server's location, the devices either communicate directly peer-to-peer or through a gateway.

One alternate system design would be to integrate Service Discovery (SD) [59] as a more flexible way for clients and servers to locate each other. In fact, there are several SD implementations (e.g., [34]) that can be integrated with the reference implementation of RICCi. One can also develop a simpler discovery component based on multicast. We plan to explore these alternate designs as a future work direction.

3.3.4 Broker Implementation

The RICCi broker system was implemented following the defined architecture presented in Section 3.2.3. In order to successfully manage messages and connected devices, the system stores each device information in `Device` objects and stores the connection information in `Session` objects. When a device makes a request, the broker system pairs it with an available second device and keeps the information regarding requester (device making data request) and provider (device providing data). All messages transmitted between connected devices have their metadata checked, in order to determine data exchange pattern, place of

origin, and destination. Depending on the information being carried, the broker will behave differently. In case of remote or stream messages, LAN broker servers transfer the local network IP from the requester device to the provider device, so that devices can use RICCi to establish the device-to-device connection. In case of copy, the broker acts as a gateway. For WAN broker servers, the broker acts as a gateway by relaying all messages between connected devices.

3.4 Motivating Scenarios Revisited

In this section, we revisit the motivating scenarios from Section 3.1 and explain how RICCi facilitates their implementation.

3.4.1 Sharing Documents Across Mobile Devices

In this scenario, users have to be able to share text documents between their mobile devices without relying on having an Internet connection. To that end, RICCi can be used to implement an application that *copies* documents between devices by building on the ICC logic for accessing files within the same device. Because RICCi mirrors the ICC interfaces, the developer would have to indicate that the documents need to be copied across the network by using the RICCi's *Copy* data exchange pattern. RICCi automatically determines which communication mode to select by analyzing the current network conditions and available resources (more details in Section 3.2.3). For example, in the absence of any Internet connectivity, the devices would talk to each other via WiFi Direct; if no WiFi Direct connection is established, but both devices are connected to the Internet, they would share the documents over a cloud-based service; finally, if both devices are configured to talk to a local

edge server, they would use it to broker their communication as well.

The issue at hand is that using RICCi enables mobile application developers to keep the same application logic for sharing the documents under any of the aforementioned network environments. Furthermore, the logic for opening a document component on another device is almost identical to that of opening a document on the same device. Any Android programmer, familiar with the ubiquitous ICC abstraction, should be able to quickly become productive when using RICCi.

3.4.2 Reading Input from IoT Devices

In this scenario, data must be streamed from IoT devices to a smartphone. To that end, developers can use RICCi to implement an application that uses the *Stream* data exchange pattern to stream the sensor data from the collecting devices to the ones displaying the information to the user. The RICCi runtime would determine which communication mode to select, based on the current network and server availability. If the user is in the vicinity of the IoT sensors (i.e., inside the house), the data would be streamed directly from device to device. If the user stepped outside into the yard, RICCi would leverage the house's edge server to mediate the communication. Finally, if the user tries to access the streamed data from a remote location, a cloud-based service would mediate the streaming session.

3.4.3 Reading Properties of Temporal Data

In this scenario, factory managers must be able to access data models on their subordinates' tablets, albeit without copying the models, which cannot be transferred across the network efficiently. To that end, the immovable collected sensor data can be represented as an Android component, exposing several remotely invocable methods that provide access to the

data's various properties. Developers can use the *Remote* data exchange pattern to remotely access these methods, which are implemented as so-called *remote callbacks*. In other words, in this case, RICCi moves the code to the data, with the provided, automatically generated query interface mirroring the methods of the Android component that encapsulates the immovable data.

3.5 Evaluation

This section describes the research questions, methodology, and experiments we conducted to evaluate RICCi.

3.5.1 Empirical Study

For the empirical evaluation, we deployed RICCi on a Google Nexus 6 and a Huawei KIW-L24 device, with the target OS Android Marshmallow 6.0.1. For power measurements, we deployed RICCi on an LG Volt LS740 and used a Monsoon power monitor[2] to gather the energy consumption information.

Our evaluation objectives are to evaluate the differences between RICCi-based and hand-coded applications in terms of their respective software engineering metrics (i.e., lines of code, complexity) and runtime performance (i.e., time and energy consumption).

Therefore, our evaluation is driven by the following two research questions:

- **RQ1**: When sharing component data across devices, how do the Software Engineering metrics of a RICCi-based solution compare to that of a hand-coded solution?

[2]https://www.msoon.com/LabEquipment/PowerMonitor/

- **RQ2**: When sharing component data across devices, how do the runtime performance and energy efficiency of a RICCi-based solution compare to that of a hand-coded solution?

The purpose of **RQ1** is to determine how a hand-coded solution compares to RICCi while performing the same set of operations, in terms of software metrics. To answer RQ1, we implemented three distributed mobile applications that exchange component data. Each application was implemented in two versions: (1) using one of RICCi data exchange patterns, (2) hand-coding the same functionality. We collected software engineering metrics for both versions of each application and then compared and contrasted them.

The rationale behind **RQ2** is to determine whether a RICCi-based or a hand-coded solution more efficient with respect to runtime performance and energy consumption. To answer RQ2, our three subject applications implement the functionality required to copy, stream, and remotely access data. For each subject, we implemented a RICCi-based and a hand-coded version. To measure the streaming operations, we used a media file of 3.64MB in size.

For copy, we implemented an application for sharing contact data. For stream, we implemented an application for streaming audio files. For remote access, we implemented an application that access methods of a `Intent` object containing contact information.

3.5.2 Evaluation Design

To measure the total number of lines of code (LoC) and code complexity, we used the Android Studio plug-in MetricsReloaded [57]. To measure runtime performance, we collected the total time taken by system operations and measured energy used by a device when executing a given component data sharing scenario. To measure energy consumption, we

Table 3.1: Metrics

Exchange Pattern	Version	LoC	v(G)avg	v(G)tot
Copy	Hand Written	4819	2.14	506
	RICCi	3730	1.9	306
Stream	Hand Written	6735	2.25	755
	RICCi	3719	1.91	303
Remote	Hand Written	7791	2.06	882
	RICCi	3874	1.83	322

Figure 3.9: Average energy consumption in Joules

used the Monsoon power monitor to collect the idle power and the average power of a LG Volt LS740, while performing each of the data exchange patterns. We then subtracted the idle power from the measured power during the experiments to determine the actual energy consumed by the distributed functionality. We calculated the energy consumption for a time interval of 70 seconds.

3.5.3 Results

Table 3.1 presents the numbers of lines of code (LoC), average cyclomatic complexity (v(G)avg), and total complexity (v(G)tot) for the total of all non-abstract methods in each project. These values show that the hand-coded versions for copy, stream, and remote are larger and more complex than their RICCi-based counterparts.

Table 3.2 shows the performance values of a RICCi-based application in terms of the time it took to complete all requests as required by different data exchange patterns under three dissimilar network environments. These results show that the time required to complete the component sharing requests is closely related to the amount of data transferred and latency of the underlying network.

Table 3.2: Average time in milliseconds to perform operation for different data exchange pattern

Network	Exchange Pattern	Total Time	Transmission Time
WAN	Copy	2867.64ms	321ms
	Stream	8183.68ms	2578.28ms
	Remote	2165.29ms	231ms
LAN	Copy	3851.64ms	306.33ms
	Stream	8522.64ms	330.66ms
	Remote	3618.36ms	326.08ms
Direct	Copy	2616.68ms	3186.43ms
	Stream	5691.56ms	2490.18ms
	Remote	3179.58ms	1204.42ms

Figure 3.9 shows the values of the average energy consumed by each of the data exchange patterns in joules. The information refers to requesters devices (devices making the requests) and providers (remote devices providing the resources). We also show average energy consumption for a basic ICC operation while retrieving contacts from one device. Figure 3.9 shows that for all cases, the average energy consumption for RICCi requesters and providers devices is lower than for operating (accessing a phone contact) using ICC 48.38J. We speculate that the higher energy consumption for ICC is related to the necessity to perform more operations than when operating within a single device. While for RICCi, the energy costs come from data transfer operations. Also, for most data exchange patterns, the provider device will be consuming more energy than the requester counterpart. The two exceptions are for the remote and stream transfers, which use direct (peer-to-peer) communication.

3.5.4 Threats to Validity

As with any study, our evaluation also carries threats to validity. We identify the following sources of validity threats.

Internal Threats: We implemented all the applications used for the experiments. It is possible that different hand-coded solutions would show different performance characteristics. Also the size of the files transferred during the experiments can impact the performance.

External Threats: In terms of power consumption, Android phones have background processes that may require additional energy. To mitigate this threat, we ended all background processes, and performed multiple runs, to properly estimate the amount of energy consumed by each test. Android OS is being continuously updated, so newer versions may be better optimized, particularly w.r.t. energy consumption, than the one that we used for our experiments, thus improving the runtime performance results across the board. In terms of time, different networks offer dissimilar transmission latencies, and different devices with dissimilar hardware capabilities can provide unequal response times. In terms of accuracy of our power measurement procedures, our readings are directly related to the accuracy of the power monitoring devices used in the experiments.

3.5.5 Summary of the Results

We derived several insights from our experimental study. First, RICCi is not only comparable to hand-coded solutions in terms of complexity and in terms of lines of code, but also can provide a smaller code size and less complex solutions for mobile application developers to maintain. Second, the runtime performance in terms of the total time can vary, being affected by the latency of the network in place and the amount of transferred data. Finally, we found that RICCi operations on each of the participating devices can consume less energy than the equivalent operations over ICC within a single device, an insight motivating the use of distributed processing to reduce the amount of energy consumed by individual devices.

3.6 Conclusions and Future Work

We have presented the design and implementation of RICCi, a distributed framework for sharing component data across Android devices. RICCi provides programming abstractions and runtime support to facilitate the implementation of inter-device component data sharing for Android applications by enhancing the Android ubiquitous ICC mechanism. The RICCi declarative programming model features a set of pre-defined data exchange patterns for developers to specify the semantics for passing component data across the network. By focusing on remote data rather than procedures, RICCi treats distributed communication latency as a first-class software design parameter. Even regarding possible security risks, the encapsulation benefits of RICCi can streamline the implementation of security enhancements. As an evaluation, we have compared the performance, complexity, and code size of RICCi-based and hand-coded versions of three distributed mobile applications that share data between applications on different Android devices. We have found that RICCi-based solutions provide comparable performance and lower code complexity than their hand-coded counterparts.

We have identified several possible future work directions. First, we would like to explore the possibility of extending RICCi to add support for heterogeneous device-to-device environments, in which, for example, Android devices can exchange components of iOS devices and vice versa. Second, we would like to thoroughly explore the security threats to RICCi-based applications and possible defenses. Third, we would like to explore how RICCi can facilitate the adaptation of distributed legacy code to be able to share component data without breaking encapsulation. And lastly, we would like to conduct controlled user studies with Android developers to determine how to improve and refine the usability and expressiveness of RICCi.

Availability

The reference implementation of RICCi as well as all the subject applications and benchmarks can be accessed via an online appendix at: https://github.com/brenodan/ricci.

Chapter 4

Case Study: Participatory Sensing

Participatory sensing [9] put forward the concept of leverage ordinary citizens for collecting and sharing sensor data from their surrounding environment by using their smartphones [62]. Participatory sensing has been shown effective in capturing environmental and behavioral information for data analysis and knowledge discovery [45]. In the traditional participatory sensing architecture, local devices collect and transfer data to a remote server for processing. Recently, the increasing volumes of participatory data, particularly the multimedia formats (i.e., photo and video), often render cloud-based processing prohibitive due to the required network transfer, which incurs high energy consumption and bandwidth utilization bottlenecks. Limited networks can be unsuitable for transferring large data volumes, as they get clogged while also draining the limited battery budgets of mobile devices.

As an alternative to dividing the data collection and processing responsibilities between local devices and remote servers, we present a novel system architecture that processes data at the edge of the network, close to where it is collected. The system leverages the computational and communication power of universally available WiFi routers. These WiFi routers coordinate the connected edge devices to cooperate on collecting and processing participatory sensing data. Specifically, each participating edge device executes a microservice to perform certain functionality, and the output of a microservice can become the input to another microservice, executed on another edge device, as coordinated by the WiFi router. Furthermore, edge devices dynamically download the executable packages of microservices

from a cloud-based repository, thus enabling our system design to perform highly dynamic participatory sensing tasks. A cloud-based coordination server controls geographically dispersed WiFi routers in different areas. In our system design, the cloud-based coordination server only needs to schedule the sensing jobs for each WiFi router, instead of controlling and communicating with individual participants directly. As a result, our design reduces the communication complexity of transferring sensing tasks, obtaining results, and handling administration, such as distributing incentive rewards.

We report on our experiences of designing, implementing, and evaluating a sensing system that constructs indoor maps by recognizing door signs. In our reference implementation for the demonstration use case, the system leverages edge-based processing to generate sensing results, verify data, retrain machine learning models, and distribute incentive rewards. Our evaluation shows that our system architecture effectively leverages the available edge resources, thereby increasing the accuracy of the sensing results, while greatly reducing network traffic. We also discuss the problems we encountered while designing and implementing our solution and then identify several open research problems that stand in the way of fully realizing the vision of edge-based participatory sensing.

The rest of this chapter is organized as follows: Section 4.1 introduces our system design and a running participatory sensing example. Section 4.2 explains our use case's implementation. Section 4.3 presents our evaluation results. Section 4.4 discusses some of the key insights derived from implementing and evaluating our solution. Section 4.5 presents concluding remarks.

Background and related work is available in Chapter 2. Chapter 6.2 discusses the applicability and limitations of our contributions. Also, the Appendix B.2 contains possible usability study directions that practitioners may follow.

4.1 System Overview

In this section, we first describe our participatory sensing system architecture, and then illustrate a typical workflow enabled by this architecture via a digital image recognition scenario.

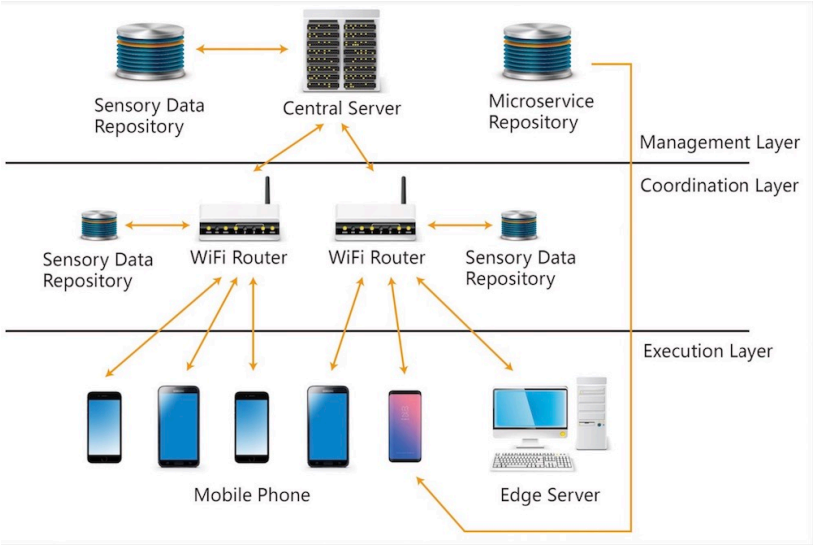

Figure 4.1: Participatory Sensing System Architecture

4.1.1 System Components

Figure 4.1 shows the main components of our system architecture, which includes a cloud-based coordination server, a cloud-based microservice and data repository, a set of WiFi routers distributed across different regions, as well as mobile devices and edge-mounted servers, connected to these routers.

This architecture is designed for task owners, individuals, or organizations that need to perform some participatory sensing task to infer some useful information. A typical workflow starts with a task owner interfacing with the cloud-based coordination server to submit a

task to perform. A participatory sensing task contains 1) a general description, including the target region, the expected number of results, and the incentive rewards for the final results; and 2) directives for executing a sequence of microservices, which collect sensor data as well as process and evaluate the data to produce the final result. A microservice encapsulates a simple functionality to execute on an edge device.

The coordination server then contacts the WiFi routers, passing them the directives. Each WiFi router coordinates a collection of surrounding devices to execute a sequence of microservices [93]. Each edge-based device downloads the execution package of its assigned microservice from the cloud-based repository, executes the microservice, returning the execution result to the WiFi router. Henceforth, we use the term "job" to refer to the procedure performed by a WiFi router to orchestrate the execution of microservices on edge devices to compute one constituent result for a task.

4.1.2 Use Case and Execution Flow

Next, we introduce a use case that demonstrates the general operation of our system architecture. Consider a large office building with an intricate floor plan, making it hard for visitors to find their way around. To address this problem, the facilities manager decides to make available an indoor navigation map. As room numbers tend to remain constant, creating the navigation map involves mapping room numbers to their occupants' names. Also, due to a high turnover rate, name-tags change frequently and continuously. Hence, for each room, she needs to obtain its number and the door sign's text, which displays the name of the room's current occupant. Instead of manually collecting the names on the door signs for all the numerous rooms in the building, she can submit a participatory sensing task to collect this information. The expected results are a set of pairs of (name, room

number). Instead of engineering a non-trivial custom participatory sensing system, one can simply connect the office building's routers to our cloud-based coordination server, which will perform all required sensing tasks.

A sensing task to obtain the required information can be processed as follows. The central coordination server obtains a list of the WiFi routers in the task's region. Based on its location, each router receives from the server a set of jobs. Executing each job generates an answer in the form of a pair (name, room number). Edge devices connected to a router cooperate to execute a job following the task's workflow graph. The workflow specifies how to collect, process, and verify the sensing data involved employing the edge devices. Hence, our system design makes it possible to reduce the amount of sensing data that needs to be transferred to the cloud. In the following discussion, we further explain how our system architecture coordinates a WiFi router and its nearby edge devices to execute a job.

4.2 Executing Jobs at the Edge

Next, we describe how our architecture orchestrates the execution of jobs by means of edge-based computing devices.

4.2.1 Work Flow Overview

Figure 4.2 shows a workflow script that describes how to execute a job. The central coordination server transfers such job scripts to individual WiFi routers. A job script uses the JSON format to provide instructions that specify how to execute a job by invoking a collection of microservices. Each microservice invocation contains the required input data, the intended output, the rules for device selection, and the next microservice. Each WiFi

```
Service: {
    task_name: DoorSignCollection,
    incentive: 100,
    number: 100,
    expiration: 2018-10-15 22:00:00,
    microservices: {
        TakePhoto: {
            device: mobile phone,
            instruction: take_photo.pdf,
            on_success: PreprocessPhoto
        },
        PreprocessPhoto: {
            device: mobile phone,
            instruction: preprocess.pdf,
            on_success: RecognizePhoto
        },
        RecognizePhoto: {
            device: mobile phone,
            instruction: recognition.pdf,
            sampling_rate: 0.3,
            low_threshold: 99,
            on_success: {
                in_sampling|threshold_trigger: VerifyPhoto
            },
        VerifyPhoto: {
            device: mobile phone,
            instruction: verification.pdf,
            on_success: {
                confirmed: exit,
                refuted: {
                    return: correction,
                    TrainModel
                }}},
        TrainModel: {
            device: edge server,
            on_success: {return: model}
            on_failure: TrainModelOnCloud
        }
        TrainModelOnCloud: {
            device: cloud server,
            on_success: {return: model}
        }}}
```

Figure 4.2: Work Flow Script

router hosts a light-weight runtime system that executes the instructions of a given job script.

4.2.2 Edge-based Data Collection and Processing

The taking photo phase involves executing a photo-taking microservice on a selected mobile phone. This phase is not automatic, as it requires the phone user to use the device's camera

to capture the image in question. To that end, a descriptive instruction message is sent to the selected mobile phone, thereby asking the user to participate in executing the photo-taking microservice. If the user agrees to participate, she can follow the sent instructions and take the photo of the assigned room's door sign. To complete the microservice's execution, the runtime system on the mobile device then uploads the taken photo to the router.

The content recognition phase involves executing two microservices on the taken photo: 1) pre-processing and 2) recognizing. The pre-processing microservice transforms an image to make it suitable to be recognized using a machine learning model. To that end, the microservice converts the raw image to grayscale and applies filters to remove noise. Then, the recognition microservice takes the pre-processed image as input and outputs the recognition result as a text string and a confidence value. The confidence value indicates the probability of the recognition result being correct. In our reference implementation, we use the OpenCV library for the image preprocessing microservice, and the TensorFlow library for the image recognition microservice.

4.2.3 Edge-based Data Verification

Data quality has always been an overriding concern in participatory sensing [80]. Errors can be introduced to the collected results for the following two reasons: 1) hardware deficiencies or software bugs; 2) participants cheating providing fake data to earn rewards. Our system design helps ensure data quality by including a human-based verification microservice into the workflow. In this use case, as getting humans into the loop is costly, in terms of both execution time and incentive rewards, the system only applies this microservice to inspect randomly picked samples and to verify the recognized results with low confidence values.

Hence, if the reported recognition confidence is lower than the user-defined threshold, or

the job is randomly selected for inspection, the human-based verification microservice is triggered. If a device owner agrees to perform the verification, she receives the same instructions as those sent to the human-assisted photo-taking microservice, as well as the actual taken photo and the corresponding recognition results; the user is asked to confirm whether the recognized result is correct for that image, and if not, what the correct result is. The returned result is used in three ways: 1) providing a verified recognition result; 2) updating the machine learning model of the image recognizing microservice; and 3) calculating the incentive rewards of the involved participants.

4.2.4 Recognition Model Update

Despite the power of machine learning-based image recognition, this technique is known to be vulnerable to errors. Hence, we use the results obtained from the data verification phase to dynamically improve the accuracy of the image recognizing microservice. If in the data verification phase, the user marks an image recognition result as wrong, while also providing the correct result, we use this information to update our image recognition model.

Our reference implementation makes use of federated learning. The original design of federated learning trains and evaluates machine learning models on distributed nodes. When a node observes an obvious improvement in recognition accuracy, it merges the model's delta with the central model base. In our system design, the router finishes all jobs assigned to it by the coordination server and feeds the pairs of images and their user-labeled content as training data to an edge server that executes the model update microservice. The microservice outputs the delta of the image recognition model and sends it to the cloud-based microservice repository.

4.2.5 Aggregated Incentive Distribution

The final phase of a job is distributing incentive rewards. Our system makes use of the hierarchical service contract strategy [39]. For each task, the coordination server establishes contracts with routers to specify the expected results and their corresponding incentive payments. When a router finishes all assigned jobs, it uploads the results to the coordination server, and receives the incentive payments. Then, for each participant, the router further calculates its payment, as guided by the contract between them. In other words, the cloud-based coordination server has no need to negotiate with each participant about the incentive payment; instead, it negotiates only with routers, which expose the execution of participatory sensing jobs as an edge service.

4.3 Evaluation

The following research questions drive our evaluation: What are the performance/energy consumption tradeoffs offered by our system architecture? What is the level of accuracy that our system architecture can achieve?

4.3.1 Reference Implementation

The specific hardware used in our experiments are as follows: a Macbook Pro laptop as the central server, the TP-Link TL-WDR3600 as the router, a ZTE Android 4.0 smart phone for computationally non-intensive edge computation, and a Lenovo Ideapad 600 laptop for computation-intensive edge processing. We flash the OpenWrt system image on the TL-WDR3600 router, so the router can be used as a regular Unix-based operating environment. Specifically, we install on it an Nginx HTTP server and a SQLite database. All devices not

only connect to the router, but also register their resource status with it.

4.3.2 Results and Analysis

Data Transmission

Table 4.1 shows the measured input and output data volumes for each microservice. In the taking photo microservice, the used photo comes from the nearby door signs, two of which appear in Figure 4.3. Pictures capturing large areas with extraneous information can be as large as 1.9 MB size. Our instructions explicitly request that users upload high-quality pictures.

Microservice	Input Size	Output Size
Taking Photo	0 KB	1.9 MB
Pre-processing Photo	1.9 MB	4 KB
Recognizing Photo	4 KB	8 bytes
Photo Verification	4 KB	8 bytes

Table 4.1: Data Transmission in Each Microservice

Figure 4.3: Experiment Photo

The pre-processing microservice converts the raw image to gray scale and applies filters to remove noise. After pre-processing, the size of the photo is compressed to 4 KB. In the photo recognition microservice, when the four digit number is successfully recognized, the result is returned as an 8-byte string. Therefore, edge-based processing requires transferring only 8 bytes of data vs. 1.9 MB that it would take to process the image in the cloud.

Recognition Latency

The recognizing photo microservice requires that the recognition model be updated whenever a new version becomes available. Due to the large size of recognition models (av. 17MB), they can take more than 7 seconds to transmit, as per our measurements. However, the time taken to download models can be used instead to recognize multiple photos using the old model. The shortened recognition time amortizes the cost of downloading an updated model. As the number of recognized photos reaches 20, the average execution time goes down to 0.68 seconds.

Microservice	**Data Downloading**	**MS Execution**	**Human Delay**	**Data Uploading**	**Total Time**
Taking Photo	0 s	0.24 s	8.12 s	0.22 s	8.58 s
Pre-processing Photo	0.23 s	1.28 s	0 s	0.1 s	1.612 s
Recognizing Photo	0.17 s	0.1 s	0 s	0.05 s	0.32 s
Verifying Photo	0.16 s	0.05 s	7.95 s	0.03 s	8.19 s

Table 4.2: Microservices' Execution Time

Energy Consumption

Since our edge server is not battery operated, we only evaluate four microservices executed on the mobile phone: 1) taking a photo, 2) pre-processing the photo, 3) recognizing the photo, and 4) verifying the photo's recognized content. We use a Monsoon power monitor to gather the energy consumption information of the mobile phone executing the microservices. Since microservices executed on the mobile phone are human-assisted, the actual time taken and energy consumed depend on human behavior. To simulate realistic human behavior, we asked three different users to execute each microservice three times and averaged the results. Table 4.2 shows the execution time of each microservice. The total energy cost of executing microservices on the mobile phone is 1574 μAh. On the contrary, if the photo

were processed by the cloud, we would only need to take a photo and upload, whose total energy consumption is 949 μAh.

Real Label	Predicted Label	Confidence
2	2	0.997632
2	2	0.997150
1	2	0.984797
3	3	0.972002

Table 4.3: Recognition Confidence

Recognition Accuracy

The experimental results show that the confidence value is high in most cases. However, Table 4.3 shows a sensing outcome that produces a misclassified digit. In this outcome, the digit 1 was incorrectly recognized as 2 due to its surrounding shadow. Since our system employs a confidence threshold procedure that verifies the correctness of the results, a human verifier was able to discover and correct the mistake, an action subsequently used to update the recognition model. Having executed the photo verification and the model retraining microservices, our system is able to correctly recognize this and similar photos. This experiment shows that the participatory sensing system eliminates as much as 99% of the data transmitted between the central server and the router, while providing the accurate recognition and reliable model updating mechanisms at the cost of additional 66% energy consumed by edge-based processing. We consider this trade-off acceptable, as the significant amount of additional energy consumed would not exhaust the mobile phone's battery, while engaging edge-based processing helps increase the accuracy of the sensing results.

4.4 Discussion

We discuss our design options and how they impact the evaluation results. Generally, in a traditional participatory sensing system, the cloud-based central server is solely responsible for both coordinating the involved edge devices and processing sensor data. We divide the responsibilities of the central server into two parts: 1) data collection and processing; 2) administrative procedures. Our design introduces two basic ideas: 1) adopt edge computing to process the sensor data near the source, to reduce the volume of sensor data being uploaded to the cloud; 2) push some parts of the administrative load onto the edge.

As demonstrated by our evaluation, one benefit of edge-based participatory sensing is the ability to push much of the data processing functionality to the edge, thereby avoiding the necessity to transfer large volumes of unprocessed data to the cloud. Besides, we also demonstrate the benefits of refining the edge-based data processing module: by updating this module incrementally, one can improve the processing accuracy with minimal networking costs. The reference implementation also demonstrates that the administrative procedures required to manage participatory sensing tasks (e.g., incentive distribution, data verification, and participant selection) can be scheduled to execute distributively on edge-based WiFi routers. Compared with traditional participatory sensing systems that schedule such procedures to execute at cloud-based servers, our system design can also reduce the network traffic and the server's computing load.

However, replacing the central server with distributed edge-based routers may introduce new problems. As the routers may overlap in their coverage ranges, some data can be collected more than once by devices connected to different routers. To avoid collecting redundant data, the participant selection procedure needs to be made aware of the overlapping coverage ranges, which presents a promising future work direction. Furthermore, our imple-

mentation relies on a fixed price incentive mechanism. Existing systems with central servers use online auctions to collect data with minimal costs. To increase sensing utility without transferring a large amount of bidding data, the system design also needs a distributed dynamic price incentive strategy.

4.5 Conclusion

As edge computing is steadily gaining prominence, researchers and practitioners alike are exploring the benefits of this computing modality for different distributed application domains. In this chapter, we have investigated the tradeoffs of leveraging edge-based processing resources for participatory sensing. We have reported on our experiences of designing, implementing, and evaluating a realistic sensing system for constructing indoor maps by recognizing door signs. What makes our system design unique is its heavy reliance on edge-based processing for a variety of tasks. Our evaluation results indicate that our system architecture can indeed leverage diverse edge-based resources, thereby reducing network traffic.

Chapter 5

Performance Estimation

5.1 Introduction

Machine learning (ML) has become an important building block for modern computing applications. As mobile, IoT, and wearable sensors collect ever-increasing volumes of data, ML models need to be continuously trained on that data. The resulting sensor data deluge renders cloud-based processing impractical, due to the privacy concerns and transfer constraints of wide-area networks. These realities give rise to edge-based ML training, as it eliminates network transmission bottlenecks and helps ensure data privacy[1]. Because edge resources are limited, edge-based ML applications must achieve high performance while utilizing resources efficiently. To that end, developers must understand how their source code translates into the performance and resource utilization of deployed edge-based ML applications [91, 96]. In this work, we focus on clustering as one of the most widely used ML techniques. We put forward a new approach that estimates an edge-based clustering application's performance and resource utilization from its source code.

The performance and resource utilization of edge applications are hard to estimate. Therefore, developers cannot easily estimate how their edge-deployed code would perform under different workloads, a particularly important issue due to the resource scarcity of edge environments [85]. Specifically, to estimate the performance of edge-based clustering, developers

[1] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5271

need to understand not only the intrinsic properties of clustering algorithms, but also the expected resource utilization trade-offs. In addition, due to cost limitations, reduced repeatability, and lack of controlled environments, developers cannot always access real test beds [35].

The design of edge-based applications can substantially impact their performance and resource consumption [63]. The design process is driven by multiple considerations, with each design choice leading to the computational power, available memory, and energy scarcity trade-offs. For example, consider designing a mobile app that employs ML for facial recognition. The choice of an ML algorithm affects not only how much processing power and memory the app would utilize, but also how much energy it would consume. Failing to recognize such trade-offs can negatively impact the user experience.

To accurately estimate an edge application's performance, developers must understand how the underlying algorithm's characteristics would affect the resulting performance and resource utilization. The most performance-affecting characteristics of the edge-application source code are computational complexity and input data size, which must be linked to performance and resource consumption (e.g., CPU usage, memory utilization, energy consumption, and task execution time) [85, 95].

Use Case: Collecting and Processing Volcanic Data

Consider a developer designing an application that collects and clusters volcanic data to predict eruptions [75]. Multiple energy harvesting devices are located at the base of a volcano to collect their region's sensor data (e.g., temperature, pressure, humidity, vibration, etc.). Each device is located at a hard-to-access location, with limited network access and energy resources. Due to resource constraints, the software and hardware stacks are optimized

to minimize memory utilization, CPU usage, and energy consumption. In addition, for budgetary reasons, it would be infeasible to set up a test bed for the application under design. Given the source code of a clustering algorithm and the estimated size of the data to be collected at the edge nodes, the developer needs to be able to estimate the energy consumption, task duration, CPU usage, memory utilization, transactions per second, and block reads per second of the application once it is deployed in the target environment.

Contributions

This chapter discusses our approach for estimating the performance of edge-based clustering applications. We describe an empirical study with an edge-based clustering application, whose four application variants differed in their algorithms (Section 5.2). We used a state-of-the-art approach for ascertaining each clustering algorithm's asymptotic complexity [99]; we identified the correlations between the collected metrics via multiple linear and polynomial regression (Section 5.3). We used the collected metrics to train a Deep Neural Network (DNN) to be able to estimate the performance and resource utilization of an ML application under design (Section 5.4). Finally, we discuss our conclusions and future work directions (Section 5.5).

Chapter 2 presents the background and related work. Chapter 6.2 discusses the applicability and limitations of our contributions. Also, the Appendix B.3 contains possible usability study directions that practitioners may follow.

The contributions of this paper are as follows:

1. We put forward a methodology for estimating the performance of edge-based ML applications.

- We empirically evaluate four popular clustering algorithms deployed at the edge, in terms of their performance and resource utilization.
- We identify how the observed performance and resource utilization correlate with each other, computational complexity, and data set size.

2. We introduce STARGAZER, a tool that reifies the methodology above.

5.2 Empirical Study

In this section, we present our empirical study's motivation, system overview, implementation, methodology, and results. The data collected in this Section is analyzed in Section 5.3 and used to train STARGAZER in Section 5.4.

5.2.1 Study Overview, Objectives, and Methodology

We deploy and benchmark the KMeans, EM, FF, and DBSC clustering algorithms in different variants of an edge-based application. We have chosen KMeans, EM, FF, and DBSC as previous works in the literature compare the performance of these algorithms [46, 84, 86, 90]. *Our study's objective is to determine how the evaluated variants differ in their respective performance & resource utilization at the edge.* In particular, we obtain the performance metrics of % CPU usage, % memory utilization, block read per second, transactions per second, energy consumption, and task duration. The privacy and security issues are considered as out of scope in this study.

Our ultimate goal is to be able to understand the impact of choosing a particular ML algorithm on the overall system performance. The measurements collected herein can inform

developers about the expected behavior of future edge-based ML applications under different requirements (see Section 5.4).

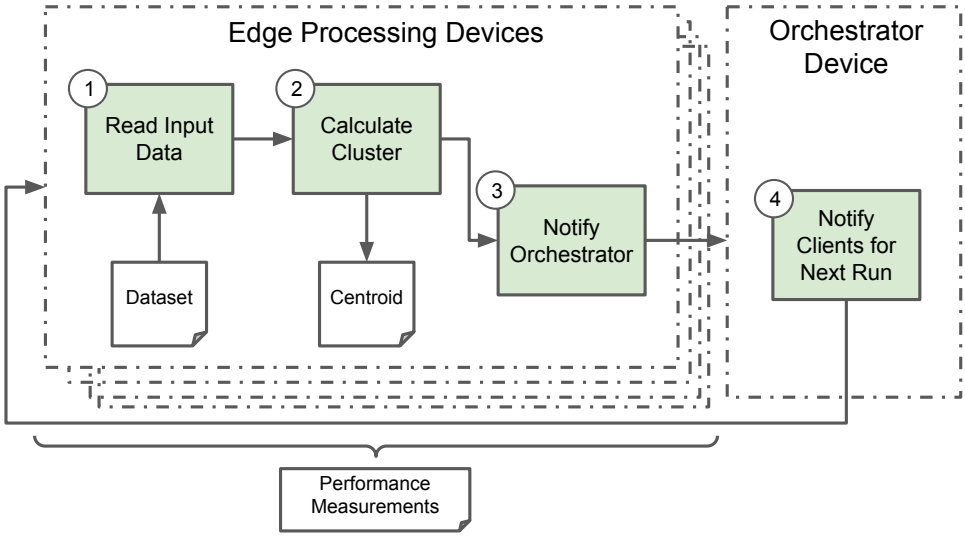

Figure 5.1: Experimental workflow on the edge.

The workflow diagram in Figure 5.1 shows our experimental edge-based clustering system that can be parameterized with any clustering algorithm. Its variants take random (x, y) coordinates as input. (**1**) Edge processing devices read their respective datasets. (**2**) Each edge processing device applies the clustering algorithm under evaluation to compute the coordinates of its unique centroid. Also, each device saves its performance and resource utilization upon completing the current run. (**3**) Each edge processing device notifies the orchestrator device that it has finished its execution. (**4**) Lastly, the orchestrator device requests that all edge processing devices perform the next run.

The study's methodology is to execute the four variants under the same load sets and collect the observed performance and resource utilization metrics. The system is composed of nine (eight edge processing devices and an orchestrator) Raspberry Pis 3 Model B, with Quad-Core 1.2 GHz and 1 GB RAM running Raspbian OS version 3.0.1. To measure energy consumption, we use a "watts up? Pro" [41]. This setup is representative of edge computing

setups performed on commodity devices[2][3].

5.2.2 System Implementation

We implemented our system in Java 1.8, the Weka library version 3.7.17 [100], and the LipeRMI [3] library for device-to-device communication. Due to our empirical study's scope, we implemented four edge processing device variants. Each variant uses a different clustering algorithm in step 2 of Figure 5.1. For our evaluation, we selected the K-Means, Farthest First, EM, and Density-Based Scan (DBSC) Weka clustering algorithms due their popularity, as a Google Scholar search shows 3,180,000 results for K-Means, 40,300 results for Farthest First, 1,150,000 results for EM, and 157,000 results for Density-Based Scan. We profiled each variant with Singularity [99] and decided to use the reported **average asymptotic complexity**, as, according to Singularity, more than one of the clustering algorithms shared the same performance values (e.g., FF and EM). The reproducibility package (Section 5.6) includes all the source code, performance measurements, and generated data.

5.2.3 Results

Figure 5.3 and 5.4 show the average % CPU usage and % memory utilization of edge processing devices respectively. We observed the highest % CPU usage value, around 23%, with `EM` algorithm while operating on input data set size 81 MB per device. We observed the highest % memory utilization, around 93%, with `EM` algorithm while operating data set size 35 MB per device. In terms of the number of transactions per second, we observed that variant `FF` the values were was considerably lower than compared to the other variants. `FF` had at most 25 transactions per second (tps) for the dataset of size 116MB, while `EM` had the

[2] https://datacenterfrontier.com/facets-of-the-edge-the-raspberry-pi-as-an-edge-ai-device/
[3] https://www.ibm.com/blogs/internet-of-things/edge-iot-analytics/

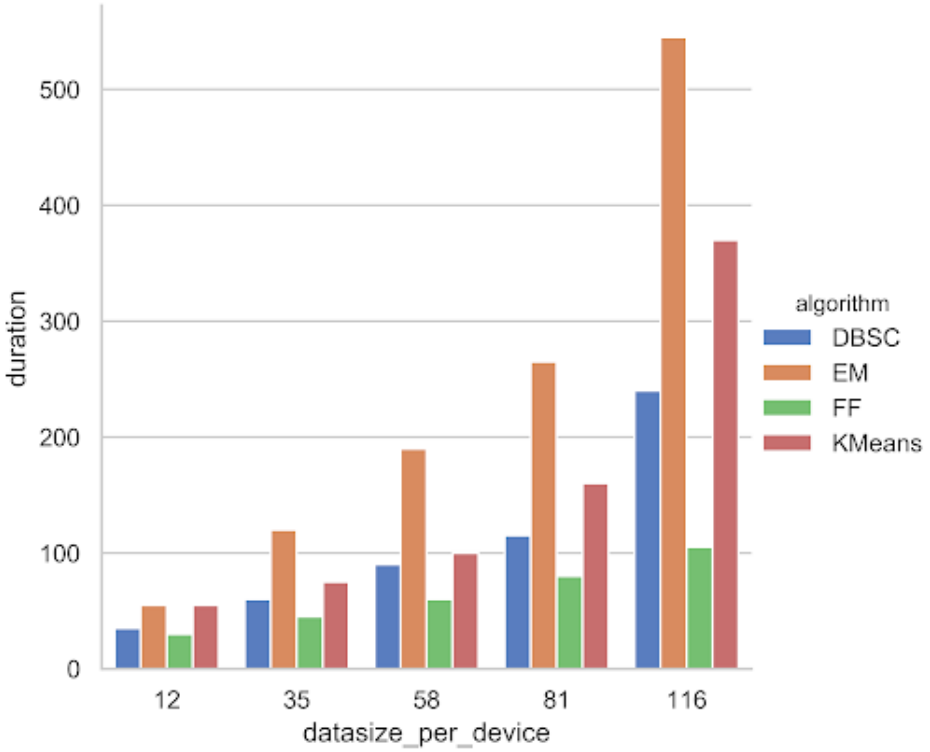

Figure 5.2: Average duration task per variant in seconds. X axis is the dataset size in MB.

highest with at more than 250 tps for the dataset of size 116MB. For energy consumption, variant `EM` had consumed on average the most joules, while variant `FF` consumed the least. We observed that the variant `FF` had the shortest execution time and that `KMeans` had the longest (see Figure 5.2), additionally, as expected, we observed a close relationship between duration and energy consumption (see Section 5.3 for more details). Regarding block reads per second (breadps), we observed that variant DBSC had the highest values with more than 6,000 breadps while operating on data set 116 MB, while the other variants had values under 3,500 for the same data set. The reproducibility package includes all graphs generated in this study.

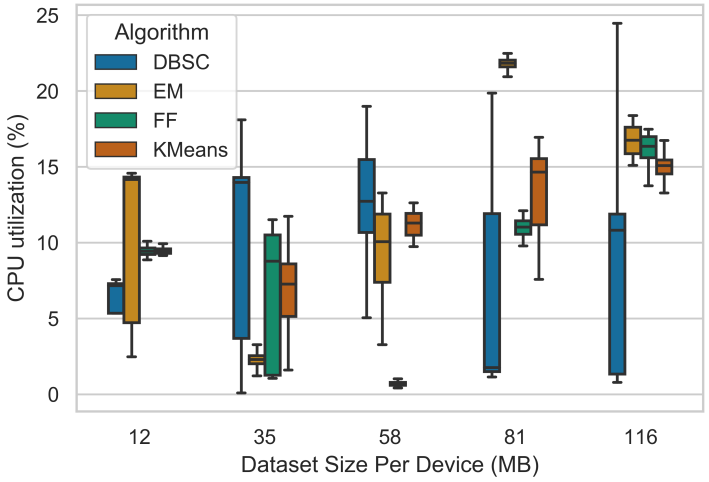

Figure 5.3: Average % CPU usage per variant. X axis is the dataset size in MB.

5.2.4 Measurements Highlights

The `EM` variant consumes more energy than the other variants on average because it makes heavier usage of the CPU and memory resources of the edge processing devices than the other variants. The `FF` variant consumes less energy and requires less time to complete its operations than the other variants, which is due to its lower asymptotic complexity.

5.2.5 Threats to Validity

Internal Threats We implemented all the systems used in our experiments. It is possible that some other implementations would have yielded different performance characteristics. Nevertheless, our implementation strategies followed widely acceptable principles by using known open-source solutions for distributed interactions.

External Threats In terms of time for completion, processes running in parallel may have influenced the results. To mitigate this threat, we terminated all background processes before running our experiments; then, we executed multiple runs, averaging the results,

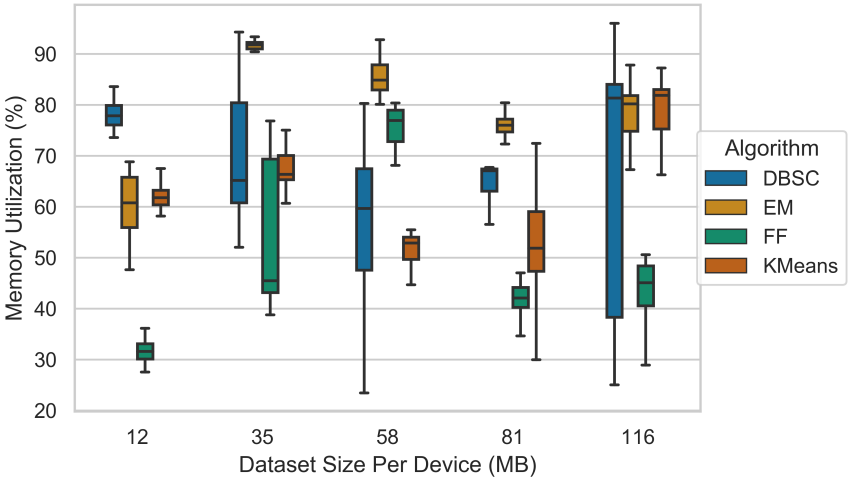

Figure 5.4: Average % memory utilization. X axis is the dataset size in MB.

so as to effectively approximate the amount of energy consumed by each test. Weka and Raspbian OS[4] are updated continuously, so newer versions may be better optimized than the one used in our experiments. In terms of time, different hardware configurations may offer dissimilar transmission latencies and response times. In terms of the accuracy of our power measurement procedures, our readings are directly related to the accuracy of the power monitoring devices used in our experiments. Finally, our findings may not be directly applicable to other clustering algorithms not covered in our evaluations. Nevertheless, if the performance of the actual clustering algorithms improves, the observed measurements are likely to change as well.

[4] https://www.raspberrypi.org/downloads/noobs/

5.3 Data Analysis

5.3.1 Motivation and Research Questions

With the increasing popularity and scale of edge-based applications, developers have to take into account a growing number of complex factors and requirements in their designs. One of the hardest system design issues is estimating how an algorithmic choice would translate into the actual performance metrics in an implemented and deployed system. To make inroads in tackling this issue, we analyse the data collected in Section 5.2 for the presence of correlations between the performance metrics, code properties, and input data size, thus seeking answers to the following questions:

- **MQ1**: Are code complexity and input data size correlated with the collected metrics?
- **MQ2**: Which collected metrics are cross-correlated?

5.3.2 Methodology

To answer **MQ1**, we developed six multilinear regression models, one for each performance metric that uses as input dataset size and algorithmic complexity. To answer **MQ2**, we apply polynomial regression models to identify whether the collected performance metrics are correlated with each other. Each model targets a different performance estimator from a given input data. We used the results from the regression analysis to determine if the dependent variables were influential in describing the variation of the dependent variable. We used root mean squared error (RMSE), F-test, and R^2 to determine the accuracy of the regression model. R^2 is a statistical measure of how close the data are to the regression line. F-test checks if the variances of two populations are the same, by comparing the

ratio of the variances. An F-test of 1 indicates that two populations are the same. The RMSE represents the differences between estimated and observed values. Additionally, it measures the model's accuracy. We used the measurements gathered from the empirical study (Section 5.2) for the regression model. Additionally, we used Singularity [99] to analyze the asymptotic complexity of the clustering algorithms. In total we used 7201 data points from the empirical study, please refer to the Availability section 5.2.2 for the data. For the regression model, we divided it into two sets. We used 80% of the data points (5761) to train the model, and we used the remaining 20% (1440) for testing. We adhere to a higher model accuracy than prior research on modeling performance in distributed applications [61]. A regression model is considered successful only if its accuracy exceeds 60%.

5.3.3 MQ1: On Complexity, Data Set Size, and Performance

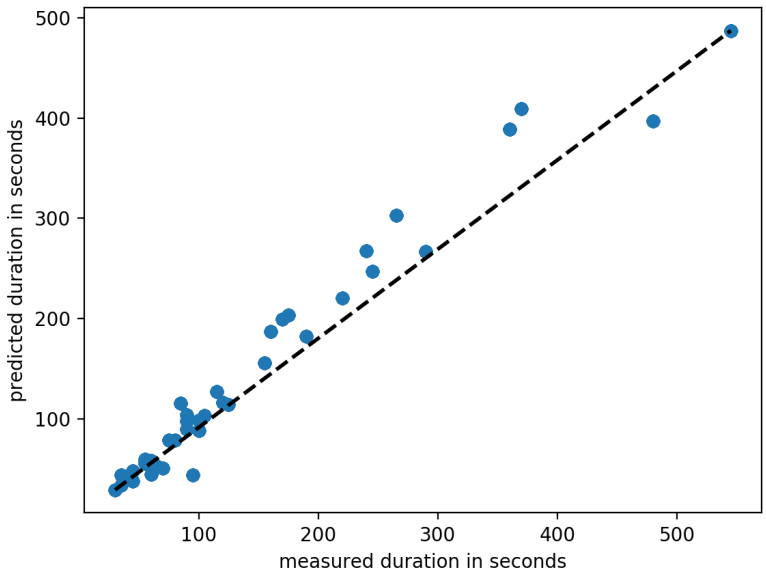

Figure 5.5: Regression line for duration.

We observed that **task duration**, and **energy consumption** are directly correlated to

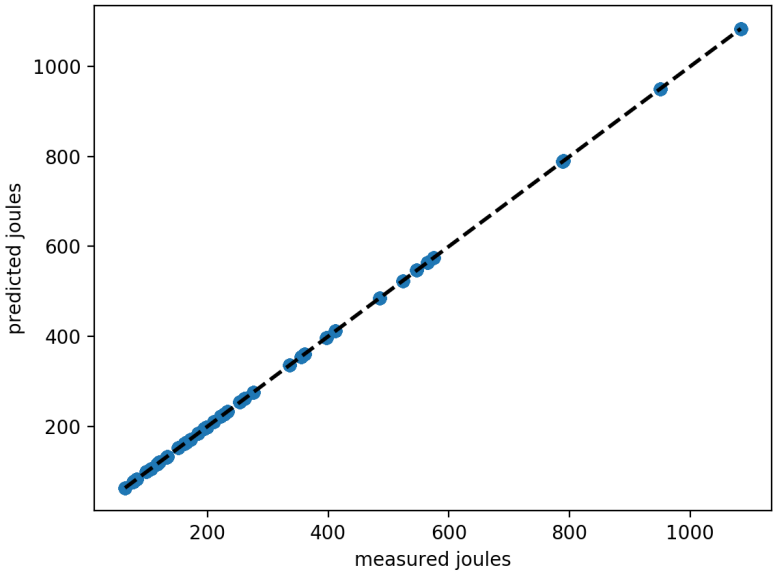

Figure 5.6: Regression line for energy consumption.

data set size per device, and **asymptotic complexity**. Figure 5.5 and Figure 5.6 show the regression lines for duration and energy consumption, respectively. We observed the following indicators for the multilinear regression models **task duration** R^2 0.95 MSE 708.50; **energy** R^2 0.97 & MSE 1857.16; We observed a small correlation between to the input data for **% CPU usage**, **%memory utilization**, **tps**, and **breadps**. We observed the following values for accuracy and mean square error: **% CPU usage** – R^2 0.33 & MSE 36.24, **%memory utilization** – R^2 0.60 & MSE 135.57, **breadps** – R^2 0.29 & MSE 1.39 Kb; tps R^2 0.34 & MSE 2384.04;

5.3.4 MQ2: Correlations Between Performance Metrics

We observed that the following correlations for performance metrics: **% memory utilization** is correlated to **energy consumption**, **duration**, **asymptotic complexity**, and **data set size**, as a 6th degree polynomial regression model has accuracy of 0.68 (R^2) and mean

square error of 111.03. We observed a correlation between **% CPU usage** and **asymptotic complexity**, **duration**, and **% memory utilization**, in 5th degree polynomial regression. We observed a accuracy of 0.84 R^2 and mean square error of 8.75.

Regarding transactions per second (**tps**), we observed the accuracy of 0.90 R^2 and mean square error of 350.69, when using a 4th degree polynomial regression model that receives as input **%memory utilization**, **asymptotic complexity**, **duration**, **%CPU usage**, and **block reads per second**. For block reads per second (**breadps**), we observed the accuracy of 0.88 R^2 and mean square error of 243.40 Bytes, when using a 4th degree polynomial regression model that receives as input **%memory utilization**, **asymptotic complexity**, **duration**, **%CPU usage**, and **tps**.

5.3.5 Summary of the Results

We developed six regression models by using the regression analysis results. We used the RMSE, F-test, and R^2 to determine the accuracy of the regression models. The developed regression models to estimate energy consumption, duration, % CPU usage, % memory utilization, transactions per second, and block reads per second are available in the reproducibility package (Section 5.6). Please refer to Appendix A.1 for additional benchmark measurements.

The following box summarizes the answers to our motivating questions:

MQ1:

- **duration** and **energy consumption** are correlated to asymptotic complexity and data set size.

MQ2:

- **% memory utilization** is correlated to data set size, duration, asymptotic complexity, and energy consumption.

- **% CPU usage** is correlated to % memory utilization, energy consumption, duration, complexity, and data set size.

- **tps** is correlated to % memory utilization, energy consumption, duration, asymptotic complexity, data set size, block reads per second, and % CPU usage.

- **breapds** is correlated to % memory utilization, energy consumption, duration, asymptotic complexity, data set size, tps, and % CPU usage.

5.4 Stargazer

Stargazer is a DNN that estimates the performance of four clustering ML algorithms. While traditional performance estimation approaches require that the code be deployed and executed, Stargazer estimates the expected performance and resource utilization of edge-based ML applications from the code properties of the given ML clustering algorithm and data input size. Developers provide the input according to how their intended system will operate, and Stargazer estimates total and per second performance of the system. That is, values for block reads per second (**breadps**), transactions per second (**tps**), percentage of memory utilization (**memory**), duration in seconds (**duration**), CPU percentage usage (**CPU**), and system energy consumption in joules.

Table 5.1: Layers of the neural network

Layer	Layer Size	Activation Function
input layer	12 neurons	Rectified Linear Unit
hidden layer #1	24 neurons	Scaled Exponential Linear Unit
hidden layer #2	36 neurons	Rectified Linear Unit
hidden layer #3	96 neurons	Exponential Linear Unit
hidden layer #4	96 neurons	Scaled Exponential Linear Unit
hidden layer #5	32 neurons	Rectified Linear Unit
hidden layer #6	24 neurons	Scaled Exponential Linear Unit
hidden layer #7	1 neurons	Linear

5.4.1 Training

We trained the Keras Sequential model [13] by minimizing the Mean-Squared-Error between the predicted and ground truth on CPUs. The stochastic gradient descent variant Adam was applied as the optimizer [50]. We use 11,955,031 data vectors, totaling 5.4GB of data to train the DNN for the estimation of a per-second performance and 7,201 data vectors for estimation of average system performance. Both training sets were collected by our empirical study (Section 5.2). We used the training/validation split of 80%/20%. When updating global models, we selected the following hyper-parameters by optimizing them throughout the empirical analysis: 128 for batch size and 200 for epochs. We supplemented these vectors with the complexity of clustering algorithms, obtained by running Singularity on the original versions of these algorithms. Table 5.1 shows the architecture of the DNN. We optimized the activation methods through trial and error, systematically experimenting with different configurations, activation methods, and numbers of neurons per layer to maximize the prediction accuracy for all metrics.

We used the insights we gained from Section 5.3 to train the DNN. To predict total task duration, we use as input data set size and algorithmic complexity. For total **energy consumption**, we use as input data set size, algorithmic complexity, and duration. For %

memory utilization prediction, we use as input data set size, complexity, and duration. For **% CPU usage** prediction, we use data set size, complexity, energy, duration, and memory. For **tps** we use as input data set size, complexity, memory, duration, energy, and breadps. For **breadps**, we use as input data set size, complexity, memory, duration, energy, and tps.

5.4.2 Results

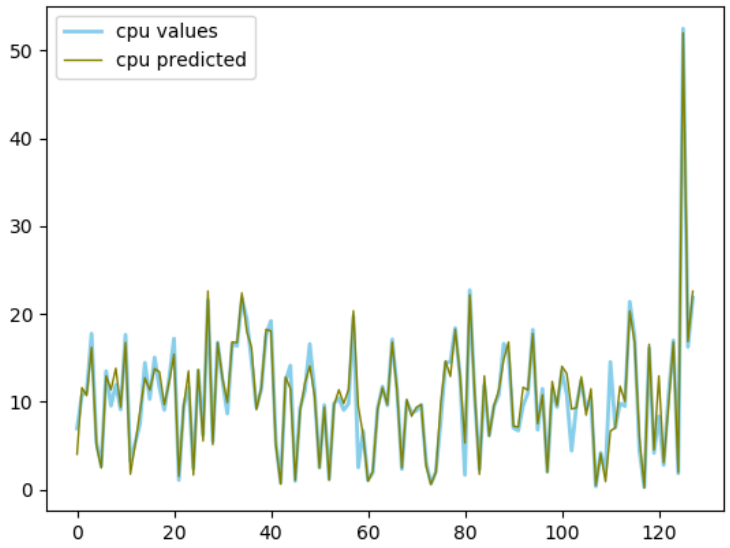

Figure 5.7: Estimated and observed system's average %CPU usage. Y axis is the % CPU usage, X is the index of the test in the validation dataset.

To estimate performance metrics in a per second arrangement, we use a training/validation split of 80% (9,564,025 data points) / 20% (2,391,006 data points), resulting in: **energy** accuracy values are 0.9999 (R^2), mean square error of 0.0340, and mean absolute error 0.28. For **total duration task** accuracy values are 0.98 (R^2), mean square error of 214.92, and mean absolute error 0.47. For **memory** accuracy values are 0.67 (R^2), mean square error of

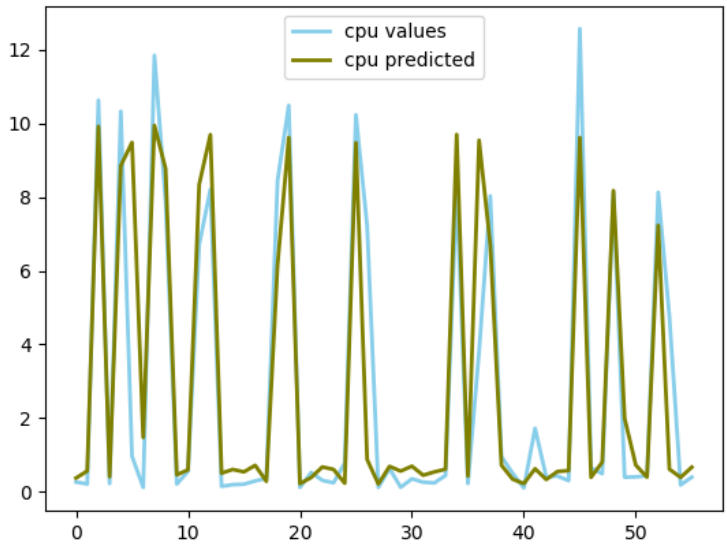

Figure 5.8: Estimated and observed system %CPU usage at a given second. Y axis is the % CPU usage, X is the index of the test in the validation dataset.

106.87, and mean absolute error 0.00. For **CPU** accuracy values are 0.89 (R^2), mean square error of 5.33, and mean absolute error 0.01. For **tps** accuracy values are 0.79 (R^2), mean square error of 326.50, and mean absolute error 0.00. For **breadps** accuracy values are 0.83 (R^2), mean square error of 197.85 Bytes, and mean absolute error 4.18. Figure 5.8 shows predicted and observed values estimating the average % CPU usage of the system at a given second.

To estimate average performance metrics, we also use a training/validation split of 80% (5,761 data points) / 20% (1,440 data points), resulting in: for **energy** estimation the accuracy values are 0.99 (R^2), mean square error of 7.98, and mean absolute error 0.14. For **total task duration** estimation the accuracy values are 0.98 (R^2), mean square error of 221.47, and mean absolute error 0.24. For **% memory utilization** estimation the accuracy values are 0.79 (R^2), mean square error of 68.34, and mean absolute error 0.00. For **% CPU usage** estimation the accuracy values are 0.98 (R^2), mean square error of 0.99, and mean

absolute error 0.01. For **tps** estimation the accuracy values are 0.88 (R^2), mean square error of 410.50, and mean absolute error 0.00. For **breadps** estimation the accuracy values are 0.98 (R^2), mean square error of 49.23 bytes, and mean absolute error 0.0. Figure 5.7 shows the estimated and the observed average % CPU usage of the system. Refer to the reproducibility package (Section 5.6) for the DNN source code and datasets.

5.4.3 Results Discussion

From operating STARGAZER, we derived the following insights. First, the STARGAZER is more accurate at estimating the application's average performance than at estimating the same metrics at a given timestamp. Second, as we identified in Section 5.3, it is impossible to accurately estimate some metrics in isolation; interconnected metrics should be evaluated in concert. For example, % CPU usage is dependent on % memory utilization.

5.4.4 Use Case Revisited

Recall the volcanic data use case in Section 5.1. Energy harvesting devices collect and locally process volcanic data. The hardware capabilities of these devices (e.g., low-power, energy harvesting, and heat dissipation) are known to application developers. In addition, based on the frequency of sensor readings, one can approximate the size of the volcanic data to be collected. However, for budgetary reasons, it would be infeasible to set up a test bed for the application under design. Assume that due to environmental temperatures, developers need to optimize the CPU usage to be under 15%, as heat dissipation is not optimal. Finally, developers know which clustering algorithm is best suited for the task. How can developers verify that once the application is deployed, its % CPU usage would be under 15%?

By using STARGAZER, developers can not only estimate the designed application's perfor-

mance and resource utilization but also identify early on whether or not it would meet the requirements. For example, developers would parameterize Stargazer with data as follows for an average system energy consumption and % CPU usage:

`Input` – **Data set size per device** → 12MB; **ML algorithm complexity**→ Singularity's Complexity Report for KMeans; **duration** → 0.27 seconds; `Output` – **average energy consumption** → 0.04 joules

`Input` – **Data set size per device** → 12MB; **ML algorithm complexity**→ Singularity's Complexity Report for KMeans; **% memory utilization**→ 50%; **duration** → 0.27 seconds; **average energy consumption** → 0.04 joules; `Output` – **% CPU usage** → 11%;

In the example above, developers learn that one device running the application would, on average, consume 0.04 joules, utilize 11% of its CPU capacity, and take 0.27 seconds to process 12MB of volcanic data. Assume that having obtained these values, developers become concerned with the possibility of the application taking too long to execute the task, as it would imply additional energy costs and CPU usage, hence more heat dissipation.

Developers would use Stargazer to estimate the performance at a timestamp, different than 0.27 seconds (e.g., 3 seconds). To that end, they parameterize Stargazer with data as follows to estimate the % CPU usage at the 3 seconds timestamp to verify if the % CPU usage will be under the 15% requirement:

`Input` – **Data set size per device** → 12MB; **ML algorithm complexity**→ Singularity's Complexity Report for KMeans; **Estimate % memory utilization**→ 50%;

time stamp → 3 seconds; **average energy consumption** → 0.04 joules; `Output` – **%CPU usage** → 7%;

As reported by Stargazer, developers would become aware that if the execution lasts for more than 3 seconds, the CPU usage is likely to reduce to 7%. This estimate shows that the designed application would meet the design constraints, thus avoiding costly deployments on a test bed.

Improving Guidelines Utility

To come up with a widely applicable and comprehensive estimation model for understanding the performance of different clustering algorithms at the edge, one would have to study numerous combinations of setups and clustering algorithms. Our study is only the first step in this process. Our hope that our findings can bootstrap many additional studies in this important area. By collating the results of such studies, one can define actionable information that would become a valuable asset in the decision-making process of system designers.

5.5 Conclusions and Future Work

In this work, we put forward Stargazer, a DNN that estimates the performance of edge-based clustering applications. To arrive at the prediction model, we performed an empirical study, in which we benchmarked four variants of an edge-based clustering application. Additionally, we used regression techniques to identify correlations between the collected metrics. We have identified several possible future work directions. First, we would like to extend our prediction models to support settings other than edge-based architectures. Second, we

would like to identify if our findings are also applicable to cloud architectures. Finally, we would like to deploy Stargazer to measure its usefulness in real-world software development settings.

5.6 Availability

The project's source code, datasets, and collected metrics are available in the following online repository: https://github.com/brenodan/stargazer.

Chapter 6

Applicability

In this chapter, we summarize our research contributions, and then discuss their applicability and limitations. This chapter can be used to help practitioners understand how to apply in practice the technologies created by the three research thrusts of this dissertation.

6.1 RICCi

In this section, we summarize our work on facilitating access to data managed by remote components. Also, we provide guidelines for Android developers to help them effectively apply our technology in their mobile applications.

6.1.1 Summary

Data-intensive applications in diverse domains, including video streaming, gaming, and health monitoring, increasingly require that mobile devices directly share data with each other. However, developing distributed data sharing functionality introduces low-level, brittle, and hard-to-maintain code into the mobile codebase. To reconcile the goals of programming convenience and performance efficiency, we created a novel middleware framework that enhances the Android platform's component model to support seamless and efficient inter-device data sharing. Our framework provides a familiar programming interface that

extends the ubiquitous Android Inter-Component Communication (ICC), thus lowering the learning curve. Unlike middleware platforms based on the RPC paradigm, our programming abstractions require that mobile application developers think through and express explicitly data transmission patterns, thus treating latency as a first-class design concern.

6.1.2 Usage Guidelines

RICCi exposes a programming model that requires mobile application developers to carefully consider how accessing remote data would impact runtime performance. Developers often lack the expertise to implement the communication logic of distributed applications. As a result, the developers often introduce low-level and hard-to-maintain code to the mobile application's codebase. The reference implementation of RICCi builds upon the Android framework to ensure a low-learning curve while handling the low-level details of accessing distributed component data. As an example, assume having to design an app that helps enforce social distancing rules. Assume that an enclosed environment is required to limit the number of simultaneous occupants. Upon entering a location, devices are required to send a copy of their randomly generated ID and position to nearby devices. Whenever the occupancy and social distancing rules are violated, the app notifies the users to move to safety. Also, the app is required to issue warnings context-specifically to its user. For example, at-risk users (e.g., above 30 BMI) are warned dissimilarly from other users. Developers may use the RICCi's *copy* transmission pattern to exchange the ID and location data.

Without RICCi, mobile application developers would have to implement every facet of the distributed interactions between the mobile devices. The additional programming effort makes the development process prone to introducing code bugs and security vulnerabilities, thus also potentially increasing maintenance costs. In contrast, with RICCi, developers can

quickly prototype and test their target application due to RICCi extending the ubiquitous and familiar ICC framework.

6.1.3 Limitations

RICCi may not be suitable for all projects. In some situations, the requirements may render this programming mechanism inapplicable. For instance, since RICCi facilitates the sharing of component data across the network, its performance is closely related to the available network's capacities. Developers deeply experienced in low-level networking and with rigid requirements in terms of security and access latencies would be advised to stay away from using distributed abstractions, such as RICCi. Instead, they can achieve comparable or even superior performance characteristics if they manually implement their own low-level, hand-optimized solutions.

6.2 Edge-based Architecture for Participatory Sensing

In this section, we present a summary of our work on edge-based architecture for participatory sensing. Also, we put forward guidelines to help system designers apply our contributions.

6.2.1 Summary

Participatory sensing collects data by means of local devices and sends it to the cloud for processing. However, transferring large volumes of the collected data to the cloud can quickly deplete the battery power of mobile devices and cause network bandwidth bottlenecks. In our work, we investigate how to engage the processing resources at the edge to

enable efficient participatory sensing with reduced network traffic. In particular, we report on the experiences of designing, implementing, and evaluating a sensing system that constructs indoor maps by recognizing door signs. A distinguishing characteristic of our system is an almost exclusive use of edge-based processing for tasks that include ML-based image recognition, human-assisted data verification, model updates, and administrative data flow aggregation. Our evaluation shows that our system architecture effectively engages the available edge resources, thereby reducing the amount of data that needs to be transferred across the network to the cloud.

6.2.2 Usage Guidelines & Application Scenarios

Our edge-based architecture for participatory sensing is designed to help developers access distributed computing resources while reducing the network traffic between the participating client devices and the cloud. Edge computing is thought as capable of reducing various bottlenecks of distributed execution. Our work explores the applicability of edge computing to the domain of participatory sensing, with the goal of removing the bottleneck caused by the necessity of transferring large volumes of sensor data across wide-area networks.

As a specific example, consider developing an earthquake detection and response app to be deployed in public spaces. This app would use the accelerometer sensor to monitor ground motions while the participating nearby devices are stationary [49]. Upon detecting an earthquake-like motion, a device would send the information to participating nearby devices for model validation. In case of an earthquake, the app alerts its users with detailed action plans. Also, participating users are rewarded with store coupons. Due to the number of users and network constraints, it would be infeasible to send all collected to cloud-based servers for processing. The sheer data volume would incur bandwidth bottlenecks, significantly

increasing the app's response time.

The above application is a typical example of participatory sensing, which has traditionally relied on the cloud for processing and storage. However, the necessity to interact with a remote cloud server for all processing and notification tasks would yield a high-latency response, making it impossible for the application to meet its timeliness requirement. In contrast, our architecture enables developers to quickly prototype the earthquake detection app by defining a set of services and their workflow script. Our evaluation demonstrates how our architecture can effectively remove the data transmission bottlenecks between the edge and the cloud (Section 4.3). Also, since our architecture takes into account the distribution of incentive rewards, users are more likely to collaborate and opt-in for the app's services.

6.2.3 Limitations

Our architecture may not be applicable to all application scenarios. In some situations, the required computational power is simply unavailable at the edge. Also, the required execution services may cause excessive energy consumption by the participating devices. This excessive energy consumption would decrease battery life, possibly reducing the number of participating users and data quality. Furthermore, our architecture requires WiFi routers for device coordination, and as such, it may be inapplicable for locations lacking those.

6.3 Stargazer

In this section, we present a summary of our contributions for estimating the performance and resource utilization of edge-based applications. We also provide guidelines to help system designers apply our methodology to solve their design problems.

6.3.1 Summary

As a solution to the sensor data deluge, edge computing processes sensor data by means of local devices. Many of these devices are resource-scarce in terms of the available processing capabilities and battery power. To achieve the required design trade-offs of edge applications, developers must be able to understand the performance and resource utilization of data processing algorithms. An increasing number of edge-based applications use machine learning (ML) as their key functionality. However, the performance and resource utilization of ML algorithms remain poorly understood, thus hindering the system design of edge-based ML applications. In addition, developers often cannot access real-world edge-based test beds during the design phase. To address this problem, we present an approach for estimating the performance of edge-based ML applications, with a particular application to clustering. To that end, we first comprehensively evaluate the performance and resource utilization of widely used clustering algorithms deployed in a representative edge environment. Second, we identify which properties of these algorithms are correlated with their performance and resource utilization. Finally, we apply our findings to create Stargazer, a Deep Neural Network that given a clustering algorithm's computational load and input data size, estimates how this algorithm would perform and utilize resources in an edge-based application. Our tool provides viable decision-making support for addressing the multifaceted design challenges of edge-based ML applications.

6.3.2 Usage Guidelines & Application Scenarios

Stargazer is designed to help system designers understand how their design choices impact the actual runtime performance of edge applications. All developers have budgetary constraints. Furthermore, many edge application designers lack access to testbeds and em-

ulators, making the development process increasingly difficult. Stargazer targets edge-based ML systems, as they may execute simple classification tasks (e.g., data clustering), which may be executed several times with differently sized datasets. The main problem that Stargazer and its methodology solve is how to provide accurate performance and resource-estimation to those designers. As a specific example, assume having to design an image recognition system that identifies birds in the wild. This system is required for wildlife researchers to be able to study the migration patterns of seabirds in the North Atlantic. Multiple energy-harvesting devices are to be placed in specific locations in the North Atlantic. The devices collect and process images to identify the bird species on their path. Also, these locations are hard-to-access, with limited network access and energy resources. Due to resource constraints, each device's execution must be optimized to minimize energy consumption. Also, due to budgetary reasons, it would be infeasible to set up a testbed for the application under design. Without our methodology, edge-based application designers would have to guess the impact of their design choices on their systems, thus wasting precious time, effort, and, possibly, preventing the deployment of their bird spotting system in the necessary time window. In contrast, with our methodology designers would arrive at a version of Stargazer which is applicable to the unique constraints of their application. In summary, our methodology relieves edge application designers from having to guess the impact of their design choices on system performance. Stargazer's value to application designers lies in its ability to estimate the performance of applications under design, so the budgetary constraints in place can be satisfied.

6.3.3 Limitations

Our methodology may be inadequate for all setups. In some situations, system designers may have access to the necessary emulators and testbeds. Also, their choice of hardware

and implementation may have stochastic properties, thus making it impossible to accurately estimate the performance and resource utilization. Furthermore, since our methodology requires that developers perform extensive data collection, it may be not applicable for projects that require a shorter time to deployment.

Chapter 7

Conclusion and Future Work

Edge computing is an emerging distributed system paradigm that has the potential to solve some of the most salient problems that stand in the way of creating next-generation mobile and IoT applications. However, developing effective and efficient edge-based applications remains hard. Because of that, developers often have to resort to ad-hoc approaches to design and implement their apps, thus introducing low-level, hard-to-maintain code to the codebase and increasing maintenance costs. The overriding goal of this dissertation research is to make the development of edge-based applications more systematic. To achieve this goal, this dissertation research was structured according to three main research thrusts: first, it designed and developed programming support and architecture that facilitate distributed data access and device-to-device communication. Second, it provided an exemplary use case for leveraging edge-based resources while reduced network bandwidth requirements. Lastly, it provided a methodology for estimating the performance and resource consumption of edge-computing applications.

We have identified several possible future work directions. First, we would like to conduct controlled user studies with Android developers to determine how to improve and refine the usability and expressiveness of RICCi. Second, we would like to thoroughly explore the security threats and possible defenses of applications, which use our architecture for edge-based participatory sensing applications. Third, we would like to explore how RICCi can facilitate the adaptation of distributed legacy code to be able to share component data without

breaking encapsulation. Fourth, we would like to explore the possibility of conducting case studies to help improve the usability and effectiveness of Stargazer. Lastly, we would like to apply the insights from Stargazer to create a methodology that allows comparing and contrasting the relative advantages of edge and cloud architectures.

Bibliography

[1] Raja Wasim Ahmad, Siti Hafizah Ab Hamid, Abdullah Gani, Mohammad S Obaidat, Junaid Shuja, Faisal Rehman, and Atta Ur Rehman Khan. Performance assessment of dynamic analysis based energy estimation tools. In *2018 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–12. IEEE, 2018.

[2] Raja Wasim Ahmad, Anjum Naveed, Joel JPC Rodrigues, Abdullah Gani, Sajjad A Madani, Junaid Shuja, Tahir Maqsood, and Sharjil Saeed. Enhancement and assessment of a code-analysis-based energy estimation framework. *IEEE Systems Journal*, 13(1):1052–1059, 2018.

[3] Felipe Santos Andrade. LipeRMI a light weight internet approach for remote method invocation, 2006. URL http://lipermi.sourceforge.net/.

[4] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308, 2014.

[5] Luciano Baresi, Luca Mottola, and Schahram Dustdar. Building software for the internet of things. *IEEE Internet Computing*, 19(2), 2015.

[6] Mikhail Bilenko, Sugato Basu, and Raymond J Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of the twenty-first international conference on Machine learning*, page 11. ACM, 2004.

[7] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[8] Nat Brown and Charlie Kindel. Distributed component object model protocol–DCOM/1.0. *Online, November*, 1998.

[9] Jeffrey A Burke, Deborah Estrin, Mark Hansen, Andrew Parker, Nithya Ramanathan, Sasank Reddy, and Mani B Srivastava. Participatory sensing. 2006.

[10] Gerardo Canfora and Fabio Melillo. Sip2Share-a middleware for mobile peer-to-peer computing. *ICSOFT*, 12:445–450, 2012.

[11] Eric Chen, Satoshi Ogata, and Keitaro Horikawa. Offloading Android applications to the cloud without customizing Android. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pages 788–793. IEEE, 2012.

[12] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. Hlscope+: Fast and accurate performance estimation for fpga hls. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 691–698. IEEE, 2017.

[13] François Chollet et al. Keras. https://keras.io, 2015.

[14] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[15] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*, 2012.

[16] B. D. Cruz, A. K. Paul, Z. Song, and E. Tilevich. Stargazer: A deep learning approach for estimating the performance of edge- based clustering applications. In *2020 IEEE International Conference on Smart Data Services (SMDS)*, pages 9–17, 2020. doi: 10.1109/SMDS49396.2020.00009.

[17] Breno Dantas Cruz. Programming support for data intensive distributed mobile applications at the edge. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 37–38, 2018.

[18] Breno Dantas Cruz and Eli Tilevich. Intent to share: Enhancing Android inter-component communication for distributed devices, 2018.

[19] Breno Dantas Cruz and Eli Tilevich. Intent to share: Enhancing android inter-component communication for distributed devices. *MobileSoft*, 2018.

[20] Breno Dantas Cruz, Junjie Cheng, Zheng Song, and Eli Tilevich. Understanding the potential of edge-based participatory sensing: an experimental study. *IEEE 91st Vehicular Technology Conference*, 2020.

[21] Li Deng, Geoffrey Hinton, and Brian Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conf. on Acoustics, Speech and Signal Processing*.

[22] Daniel J Dubois, Yosuke Bando, Konosuke Watanabe, and Henry Holtzman. ShAir: Extensible middleware for mobile peer-to-peer resource sharing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 687–690. ACM, 2013.

[23] Jeffrey R Edwards and Mark E Parry. On the use of polynomial regression equations as an alternative to difference scores in organizational research. *Academy of Management journal*, 36(6):1577–1613, 1993.

[24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[25] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[26] Huber Flores and Satish Srirama. Mobile code offloading: should it be a local decision or global inference? In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 539–540. ACM, 2013.

[27] Huber Flores, Pan Hui, Sasu Tarkoma, Yong Li, Satish Srirama, and Rajkumar Buyya. Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, 53(3):80–88, 2015.

[28] Jacob Gabrielson. Challenges with distributed systems), 2020. URL https://aws.amazon.com/builders-library/challenges-with-distributed-systems/.

[29] Stylianos Gisdakis, Thanassis Giannetsos, and Panos Papadimitratos. Shield: A data verification framework for participatory sensing systems. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 16. ACM, 2015.

[30] Google. Android interface definition language (aidl), 2011. URL http://developer.android.com/guide/developing/tools/aidl.html.

[31] Google. Binder Java documentation, 2011. URL http://developer.android.com/reference/android/os/Binder.html.

[32] Google. gRPC a high performance, open-source universal rpc framework, 2017. URL https://grpc.io.

[33] Alexander Grebhahn, Norbert Siegmund, Harald Köstler, and Sven Apel. Performance

prediction of multigrid-solver configurations. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 69–88. Springer, 2016.

[34] IETF Zeroconf Working Group. Zero configuration networking (zeroconf), 1999. URL http://www.zeroconf.org/.

[35] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[36] Karim Habak, Mostafa Ammar, Khaled A Harras, and Ellen Zegura. Femto clouds: Leveraging mobile devices to provide cloud service at the edge. In *2015 IEEE 8th international conference on cloud computing*, pages 9–16. IEEE, 2015.

[37] Muhammad Habib ur Rehman, Prem Prakash Jayaraman, Mohamed Medhat Gaber, et al. Rededge: A novel architecture for big data processing in mobile edge computing environments. *Journal of Sensor and Actuator Networks*, 6(3):17, 2017.

[38] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[39] Irfan Ul Haq, Altaf Ahmad Huqqani, and Erich Schikuta. Hierarchical aggregation of service level agreements. *Data & Knowledge Engineering*, 70(5):435–447, 2011.

[40] Najmul Hassan, Saira Gillani, Ejaz Ahmed, Ibrar Yaqoob, and Muhammad Imran. The role of edge computing in internet of things. *IEEE Communications Magazine*, 56(11):110–115, 2018.

[41] Jason M Hirst, Jonathan R Miller, Brent A Kaplan, and Derek D Reed. Watts up? pro ac power meter for automated energy recording, 2013.

[42] Nikhil Jain, Abhinav Bhatele, Michael P Robson, Todd Gamblin, and Laxmikant V Kale. Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 95. ACM, 2013.

[43] RMI Javasoft Java. Java remote method invocation specification. *Sun Microsystems*, 1997.

[44] Devki Nandan Jha, Khaled Alwasel, Areeb Alshoshan, Xianghua Huang, Ranesh Kumar Naha, Sudheer Kumar Battula, Saurabh Garg, Deepak Puthal, Philip James, Albert Y Zomaya, et al. Iotsim-edge: A simulation framework for modeling the behaviour of iot and edge computing environments. *arXiv preprint arXiv:1910.03026*, 2019.

[45] Salil S Kanhere. Participatory sensing: Crowdsourcing data from mobile smartphones in urban spaces. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 2, pages 3–6. IEEE, 2011.

[46] Adem Karahoca and Ali Kara. Comparing clustering techniques for telecom churn management. In *Proceedings of the 5th WSEAS International Conference on Telecommunications and Informatics*, pages 27–29, 2006.

[47] Samuel C Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. 1994.

[48] F Khademi and K Behfarnia. Evaluation of concrete compressive strength using artificial neural network and multiple linear regression models. *Iran University of Science & Technology*, 6(3):423–432, 2016.

[49] Irshad Khan. Collaborative earthquake detection and response using smart devices.

In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 36–37, 2020.

[50] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[51] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[52] Kamran Kowsari, Donald E Brown, Mojtaba Heidarysafa, Kiana Jafari Meimandi, Matthew S Gerber, and Laura E Barnes. Hdltex: Hierarchical deep learning for text classification. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 364–371. IEEE, 2017.

[53] Hans-Peter Kriegel, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(3): 231–240, 2011.

[54] Young-Woo Kwon and Eli Tilevich. Energy-efficient and fault-tolerant distributed mobile execution. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 586–595. IEEE, 2012.

[55] Young-Woo Kwon and Eli Tilevich. The impact of distributed programming abstractions on application energy consumption. *Information and Software Technology*, 55 (9):1602–1613, 2013.

[56] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436–444, 2015.

[57] Bas Leijdekkers. Android interface definition language (AIDL), 2016. URL https://plugins.jetbrains.com/plugin/93-metricsreloaded.

[58] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTa: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[59] Mikael Linden, David Simonsen, Andreas Åkre Solberg, Ingrid Melve, and Walter M Tveter. Kalmar union, a confederation of nordic identity federations. In *TERENA Networking Conference*, 2009.

[60] Guangqun Liu, Yan Xu, Zongjiang He, Yanyi Rao, Junjuan Xia, and Liseng Fan. Deep learning-based channel prediction for edge computing networks toward intelligent connected vehicles. *IEEE Access*, 7:114487–114495, 2019.

[61] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas. Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 73–80. IEEE, 2012.

[62] Huadong Ma, Dong Zhao, and Peiyan Yuan. Opportunities in mobile crowd sensing. *IEEE Communications Magazine*, 52(8):29–35, 2014.

[63] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[64] James MacQueen et al. Some methods for classification and analysis of multivariate

observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[65] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[66] Martina Marjanović, Aleksandar Antonić, and Ivana Podnar Žarko. Edge computing architecture for mobile crowdsensing. *IEEE Access*, 6:10662–10674, 2018.

[67] Giuseppe Massari, Michele Zanella, and William Fornaciari. Towards distributed mobile computing. In *Mobile System Technologies Workshop (MST), 2016*, pages 29–35. IEEE, 2016.

[68] Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[69] Min Mun, Sasank Reddy, Katie Shilton, Nathan Yau, Jeff Burke, Deborah Estrin, Mark Hansen, Eric Howard, Ruth West, and Péter Boda. Peir, the personal environmental impact report, as a platform for participatory sensing systems research. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 55–68. ACM, 2009.

[70] Kazuhiro Nakao and Yukikazu Nakamoto. Toward remote service invocation in Android. In *9th International Conference on Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC)*, pages 612–617. IEEE, 2012.

[71] B J Nelson. *Remote procedure call*. PhD thesis, Pittsburgh Univ., Pittsburgh, PA, 1981. URL http://cds.cern.ch/record/132187. Presented on May 1981.

[72] Milad Zafar Nezhad, Dongxiao Zhu, Najibesadat Sadati, and Kai Yang. A predictive approach using deep feature learning for electronic medical records: A comparative study. *arXiv preprint arXiv:1801.02961*, 2018.

[73] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[74] Sangeun Oh, Hyuck Yoo, Dae R Jeong, Duc Hoang Bui, and Insik Shin. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. 2017.

[75] Justin Parra, Olac Fuentes, Elizabeth Anthony, and Vladik Kreinovich. Use of machine learning to analyze and–hopefully–predict volcano activity. *Acta Polytechnica Hungarica*, 14(3), 2017.

[76] Anna-Kaisa Pietiläinen, Earl Oliver, Jason LeBrun, George Varghese, and Christophe Diot. MobiClique: middleware for mobile social networking. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 49–54. ACM, 2009.

[77] Kevin Pinte, Dries Harnie, and Theo D'Hondt. Enabling cross-technology mobile applications with network-aware references. In *Coordination Models and Languages*, pages 142–156. Springer, 2011.

[78] Pierluigi Plebani, Cinzia Cappiello, Marco Comuzzi, Barbara Pernici, and Sandeep Yadav. MicroMAIS: executing and orchestrating web services on constrained mobile devices. *Software: Practice and Experience*, 42(9):1075–1094, 2012.

[79] Dave Raggett. The web of things: Challenges and opportunities. *Computer*, 48(5): 26–32, 2015.

[80] Sasank Reddy, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. A framework for data quality and feedback in participatory sensing. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 417–418. ACM, 2007.

[81] Sasank Reddy, Andrew Parker, Josh Hyman, Jeff Burke, Deborah Estrin, and Mark Hansen. Image browsing, processing, and clustering for participatory sensing: lessons from a dietsense prototype. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 13–17. ACM, 2007.

[82] Pei Ren, Xiuquan Qiao, Junliang Chen, and Schahram Dustdar. Mobile edge computing–a booster for the practical provisioning approach of web-based augmented reality. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 349–350. IEEE, 2018.

[83] Francesco Restuccia, Sajal K Das, and Jamie Payton. Incentive mechanisms for participatory sensing: Survey and research challenges. *ACM Transactions on Sensor Networks (TOSN)*, 12(2):13, 2016.

[84] S Revathi and T Nalini. Performance comparison of various clustering algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(2), 2013.

[85] Juan Manuel Rodriguez, Cristian Mateos, and Alejandro Zunino. Energy-efficient job stealing for cpu-intensive processing in mobile devices. *Computing*, 96(2):87–117, 2014.

[86] IP Rozhnov, VI Orlov, and LA Kazakovtsev. Ensembles of clustering algorithms for problem of detection of homogeneous production batches of semiconductor devices. 2018.

[87] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for VM-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[88] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[89] Thorsten Schreiber. Android binder. *A shorter, more general work, but good for an overview of Binder. http://www .nds .rub .de/media /attachments /files/2012/03/binder.pdf*, 2011.

[90] Garima Sehgal and Dr Kanwal Garg. Comparison of various clustering algorithms. *International Journal of Computer Science and Information Technologies*, 5(3):3074–3076, 2014.

[91] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[92] Magnus Skjegstad, Frank T Johnsen, Trude H Bloebaum, and Torleiv Maseng. Mist: A reliable and delay-tolerant publish/subscribe solution for dynamic networks. In *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*, pages 1–8. IEEE, 2012.

[93] Zheng Song and Eli Tilevich. PMDC: Programmable mobile device clouds for convenient and efficient service provisioning. In *IEEE International Conference on Cloud Computing (IEEE Cloud)*. IEEE, 2018.

[94] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1:1–1:40, August

2009. ISSN 1049-331X. doi: 10.1145/1555392.1555394. URL http://doi.acm.org/10.1145/1555392.1555394.

[95] Asfa Toor, Saif ul Islam, Ghufran Ahmed, Sohail Jabbar, Shehzad Khalid, and Abdullahi Mohamud Sharif. Energy efficient edge-of-things. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):82, 2019.

[96] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26. IEEE, 2016.

[97] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications mag.*, 35(2):46–55, 1997.

[98] Steve Vinoski. RPC under fire. *IEEE Internet Computing*, 9(5):93–95, 2005.

[99] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 213–223. ACM, 2018.

[100] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[101] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the java™ system. *Computing Systems*, 9:265–290, 1996.

[102] Zhenyu Zhou, Haijun Liao, Bo Gu, Kazi Mohammed Saidul Huq, Shahid Mumtaz, and Jonathan Rodriguez. Robust mobile crowd sensing: When deep learning meets edge computing. *IEEE Network*, 32(4):54–60, 2018.

Appendices

Appendix A

Supplementary Materials

A.1 Stargazer: Performance Measurements

In this section, we present additional measurements from the STARGAZER's empirical study 5.2. These measurements are from devices that performed the clustering tasks (**clients**) and from the **server**, which orchestrated the clients.

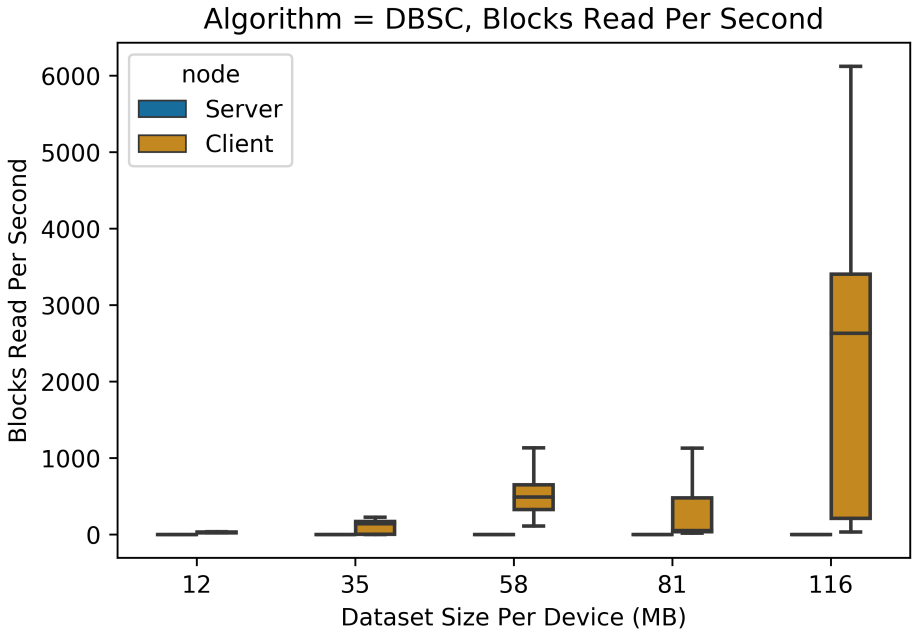

Figure A.1: Measurements for block reads per second of client and orchestrating devices executing DBSC clustering algorithm

Figures A.45, A.46, A.47, and A.48 shows measurements for block reads per second

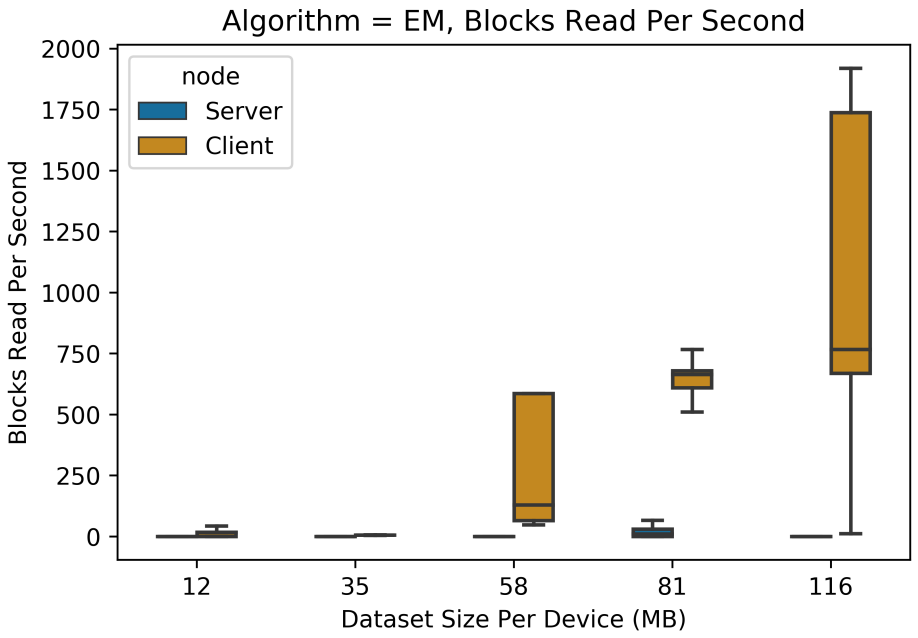

Figure A.2: Measurements for block reads per second of client and orchestrating devices executing EM clustering algorithm

(breadps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). The DBSC clustering algorithm on average performs more block-read operations than the other algorithms. The EM had the lowest number of block reads per second on average.

Figures A.5, A.6, A.7, and A.8 shows measurements for block writes per second (bwrites) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). The DBSC algorithm on average performs the highest number of block write operations per second. The FF algorithm had the lowest number of block write operations per second.

Figures A.9, A.10, A.11, and A.12 shows measurements for % CPU utilization of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). The EM algorithm had the highest average CPU utilization. The FF algorithm

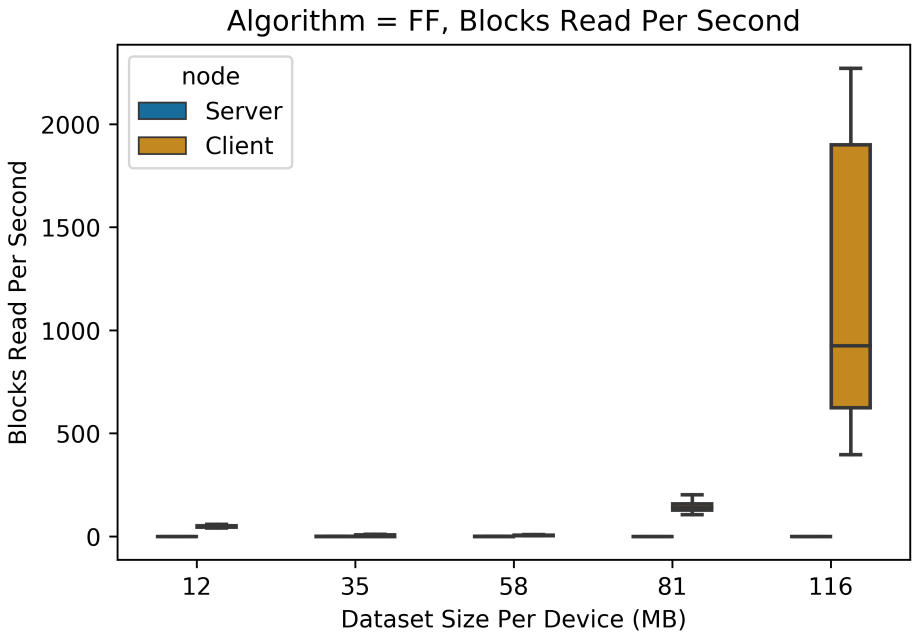

Figure A.3: Measurements for block reads per second of client and orchestrating devices executing FF clustering algorithm

had the lowest average CPU utilization.

Figures A.13, A.14, A.15, and A.16 shows measurements for % memory utilization of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed the highest average memory utilization on the EM while operation over the dataset of 35 MB. We observed the lowest memory utilization on the FF algorithm while operating over the dataset of 12 MB.

Figures A.17, A.18, A.19, and A.20 shows measurements for data receiving rate (rxkbps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observer the DBSC algorithm had the highest data receiving rate on data 35 MB. We observed that devices operating the EM algorithm had the lowest data receiving rate while operating over dataset 35 MB. On average the orchestrator device had the highest data receiving rate for the experiments of the FF algorithm while operating over

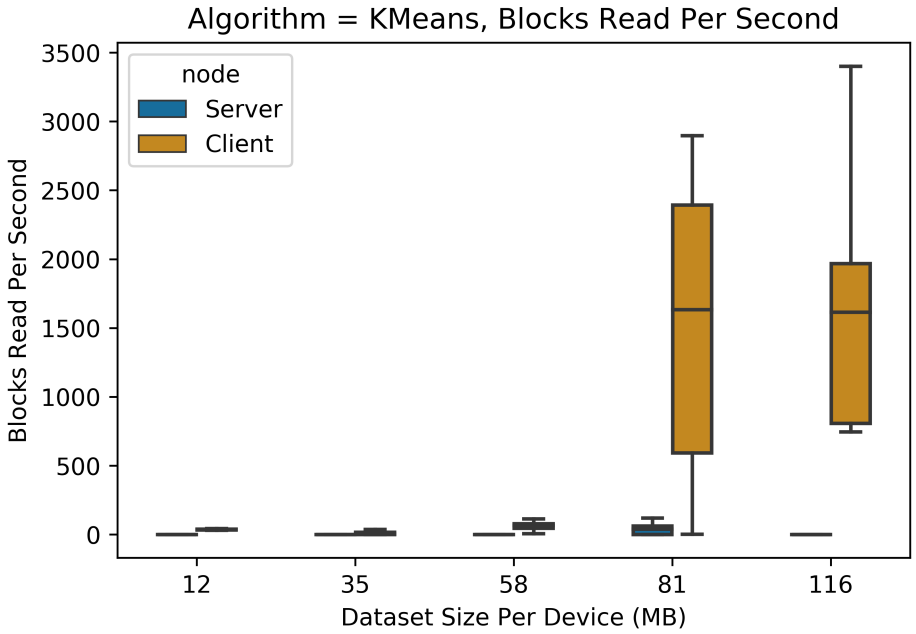

Figure A.4: Measurements for block reads per second of client and orchestrating devices executing KMeans clustering algorithm

the dataset of 12MB.

Figures A.21, A.22, A.23, and A.24 shows measurements for packet receiving rate (rxpckps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). The FF algorithm had the highest packet receiving rate while operating on the dataset of 35 MB. The FF algorithm also had the lowest while operating over dataset 116 MB.

Figures A.25, A.26, A.27, and A.28 shows measurements for transactions per second (tps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed that the DBSC algorithm had the highest number of transactions per second while operating over the dataset 116 MB. We observed that the FF algorithm displayed the lowest number of transactions per second for the dataset 12MB.

Figures A.29, A.30, A.31, and A.32 shows measurements for transactions per second (rtps)

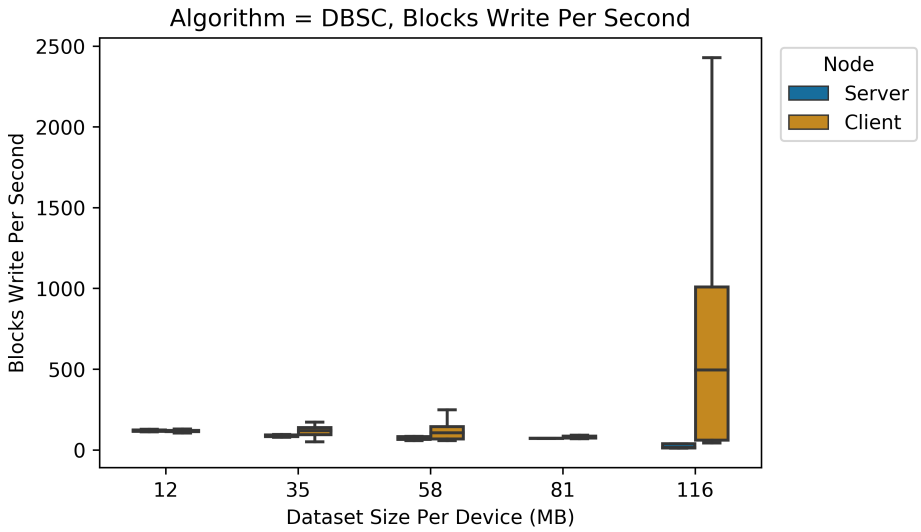

Figure A.5: Measurements for block writes per second of client and orchestrating devices executing DBSC clustering algorithm

of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed that the DBSC algorithm had the highest number of read transactions per second for the dataset 116MB. The FF algorithm had the lowest number of read transactions per second for the dataset 35 MB.

Figures A.33, A.34, A.35, and A.36 shows measurements for write transactions per second (wtps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed that the KMeans algorithm had the highest values for the write transactions per second for dataset 116 MB. The FF algorithm displayed considerably lower values for write transactions per second than the other algorithms.

Figures A.37, A.38, A.39, and A.40 shows measurements for average data transmitting rate (txkbps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed the highest txkbps on devices operating the FF algorithm on the dataset of 12 MB. We observed the lowest txkbps on devices operating the KMeans algorithm on the dataset 116 MB.

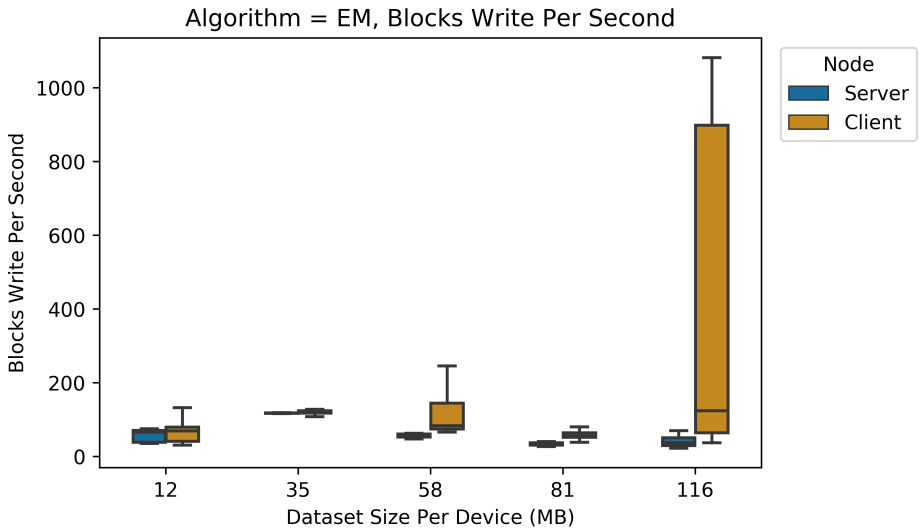

Figure A.6: Measurements for block writes per second of client and orchestrating devices executing EM clustering algorithm

Figures A.41, A.42, A.43, and A.44 shows measurements for average packet transmitting rate (txpckps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed that the highest txpckps on devices operating the FF algorithm for the dataset of 35 MB. We observed the lowest txpckps for devices operating the KMeans algorithm.

Figures A.45, A.46, A.47, and A.48 shows measurements for block reads per second (breadps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed that the highest breadps on devices operating the DBSC algorithm for the dataset of 116 MB. We observed the lowest breadps for devices operating the KMeans algorithm on dataset 35 MB.

Figures A.49, A.50, A.51, and A.52 shows measurements for block writes per second (bwriteps) of devices executing DBSC, EM, FF, and KMeans algorithms (clients) and orchestrating devices (server). We observed the highest bwriteps on devices operating the DBSC algorithm for the dataset 116MB. We observed the lowest bwriteps on devices oper-

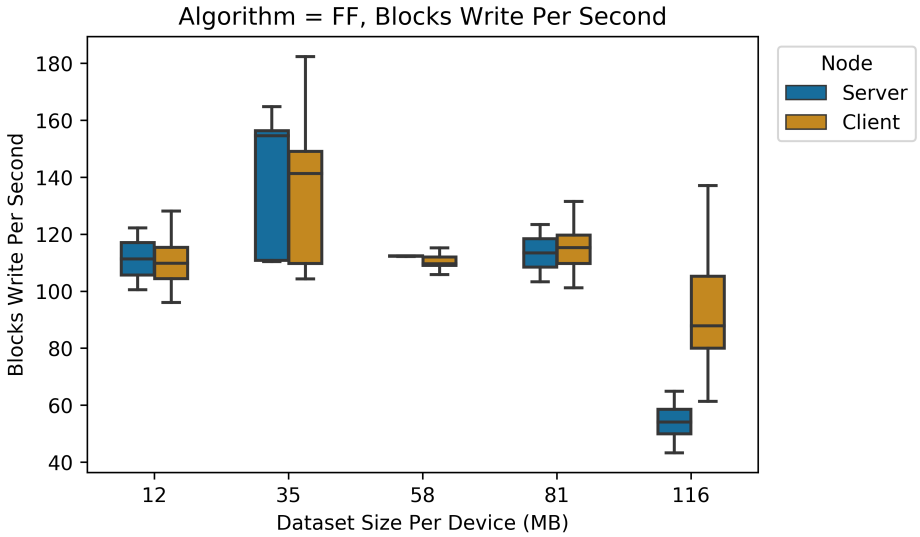

Figure A.7: Measurements for block writes per second of client and orchestrating devices executing FF clustering algorithm

ating the FF on the dataset 116MB.

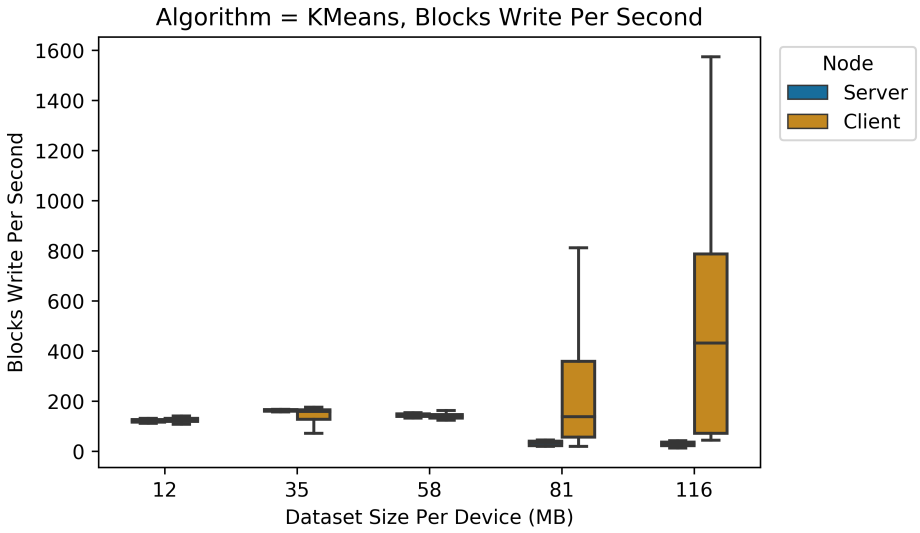

Figure A.8: Measurements for block writes per second of client and orchestrating devices executing KMeans clustering algorithm

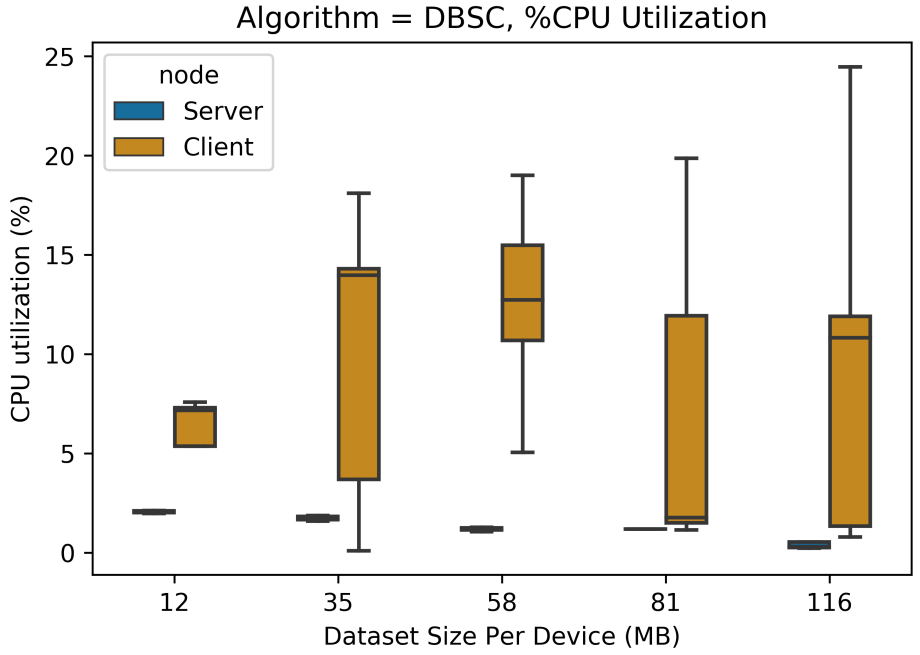

Figure A.9: Measurements for average CPU utilization of client and orchestrating devices executing DBSC clustering algorithm

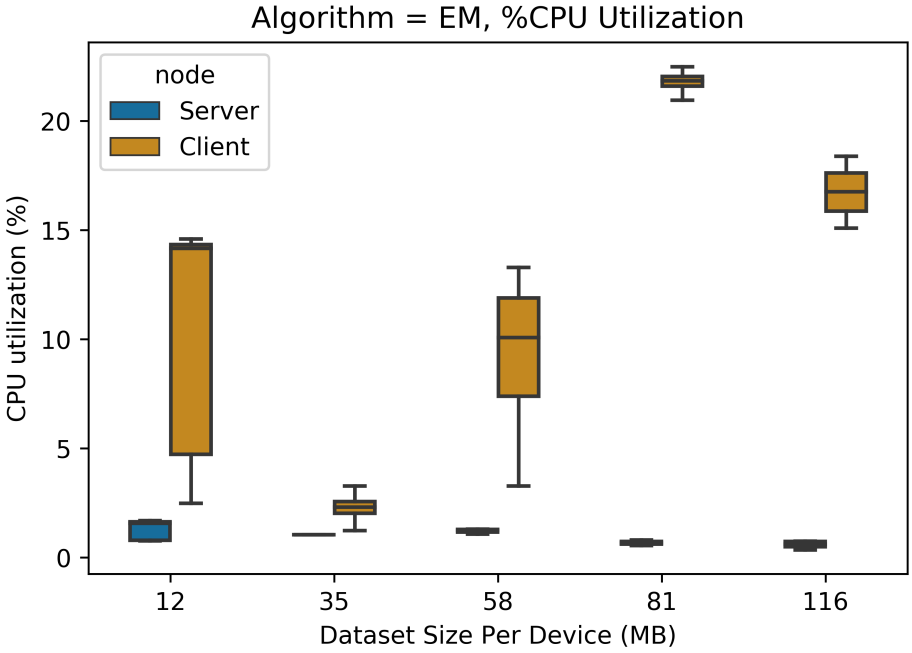

Figure A.10: Measurements for average CPU utilization of client and orchestrating devices executing EM clustering algorithm

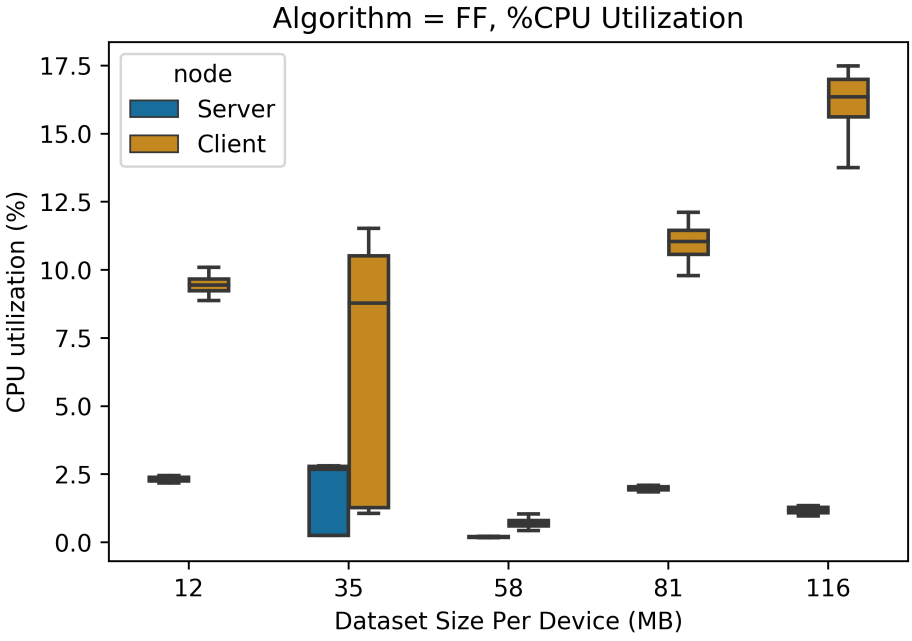

Figure A.11: Measurements for average CPU utilization of client and orchestrating devices executing FF clustering algorithm

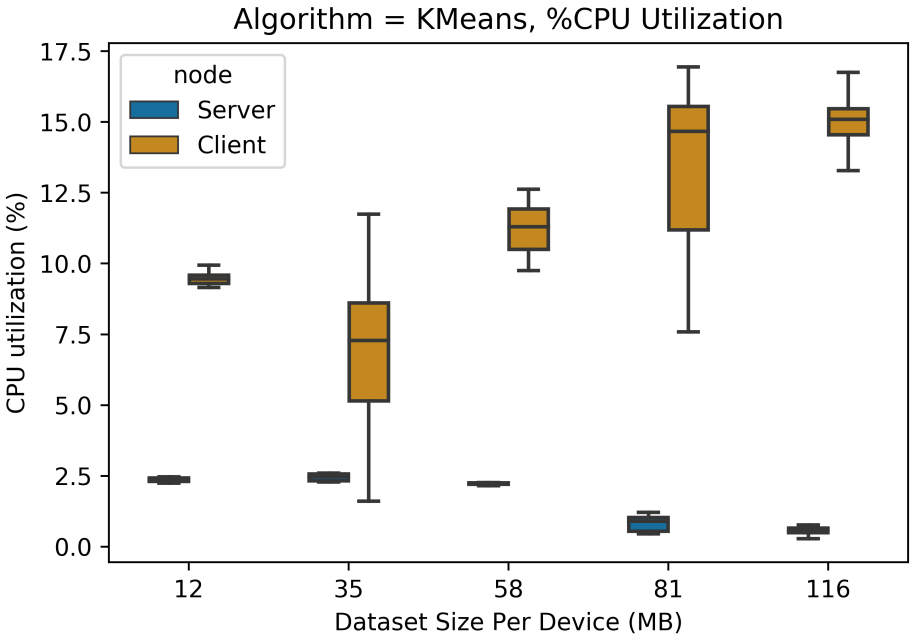

Figure A.12: Measurements for average CPU utilization of client and orchestrating devices executing KMeans clustering algorithm

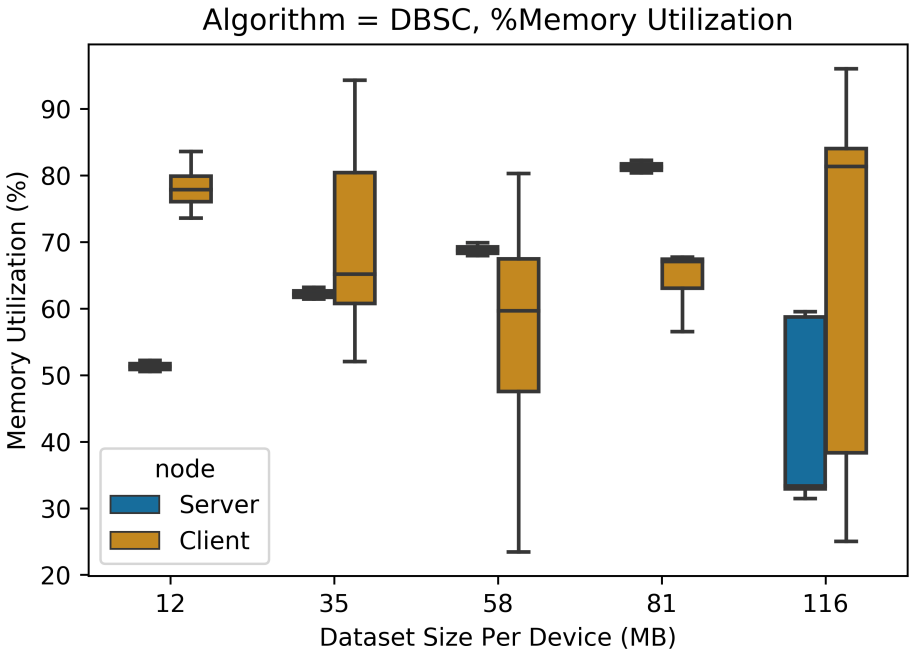

Figure A.13: Measurements for average memory utilization of client and orchestrating devices executing DBSC clustering algorithm

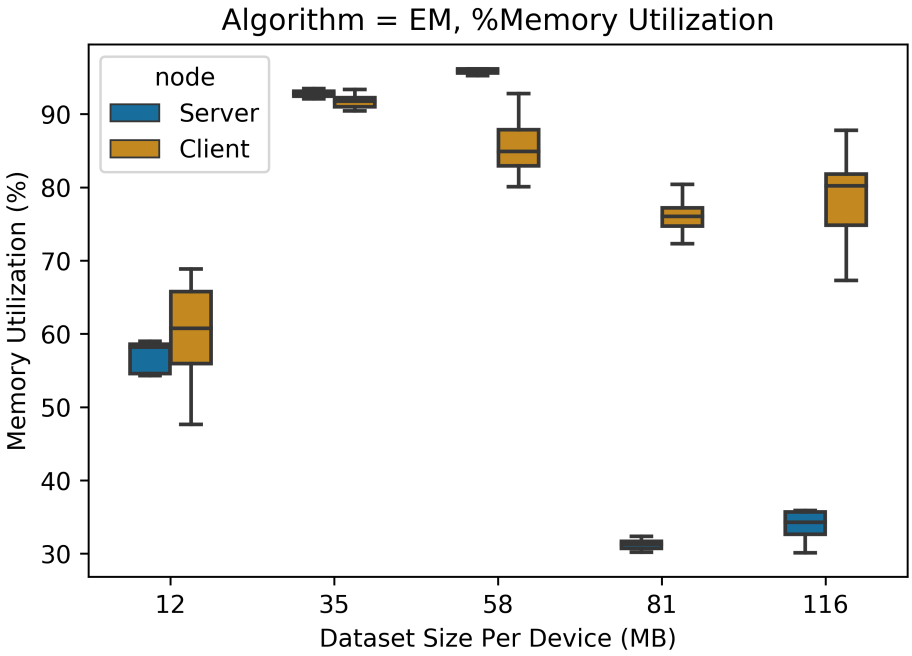

Figure A.14: Measurements for average memory utilization of client and orchestrating devices executing EM clustering algorithm

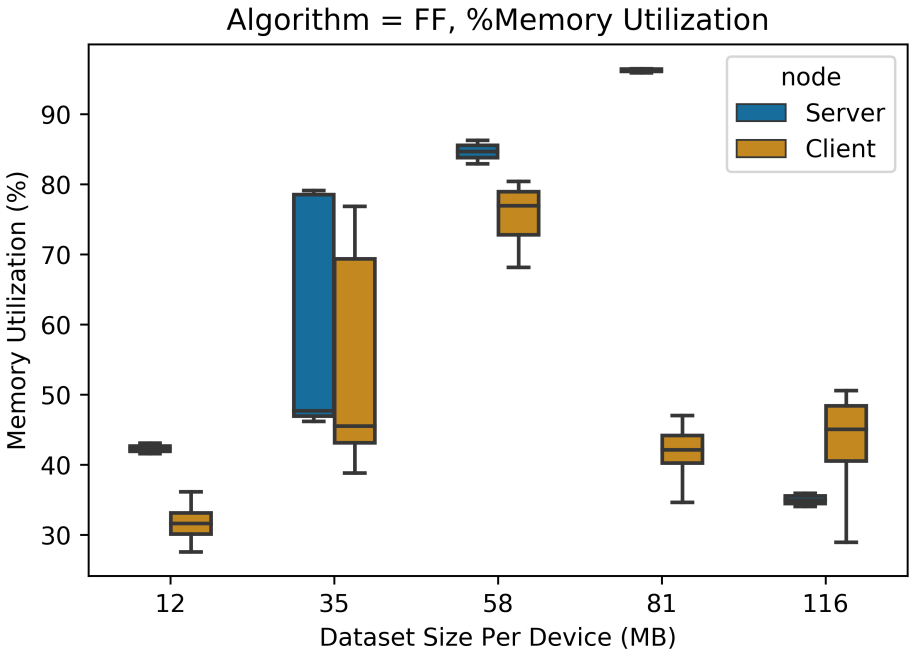

Figure A.15: Measurements for average memory utilization of client and orchestrating devices executing FF clustering algorithm

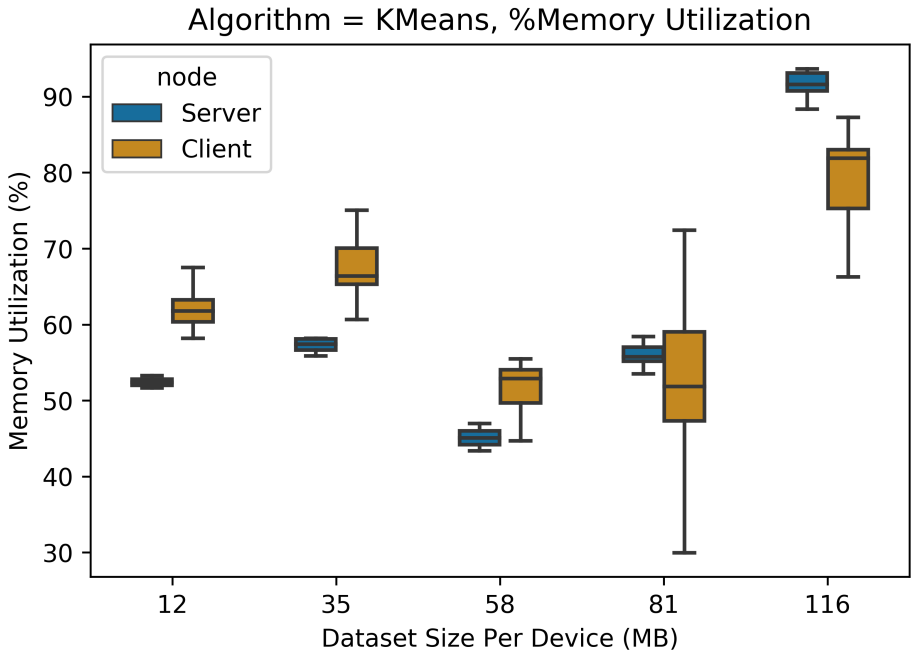

Figure A.16: Measurements for average memory utilization of client and orchestrating devices executing KMeans clustering algorithm

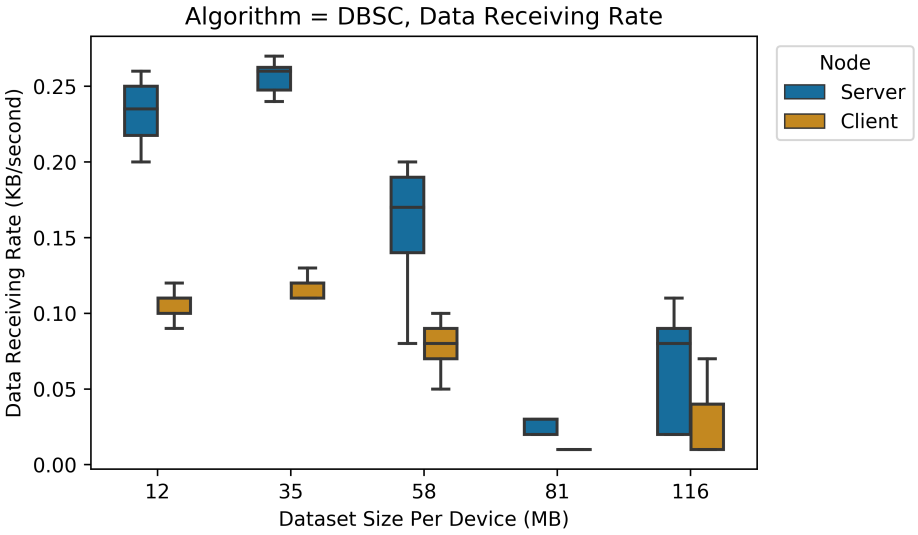

Figure A.17: Measurements for average data receive rate of client and orchestrating devices executing DBSC clustering algorithm

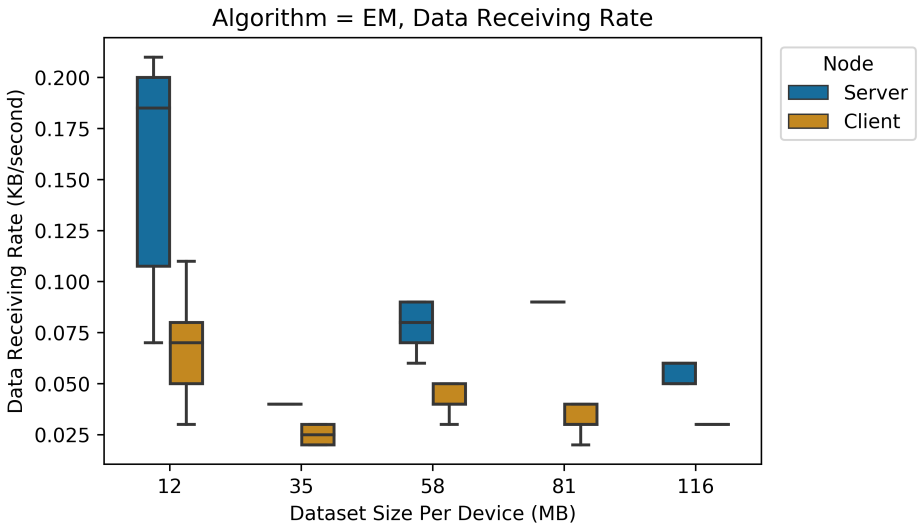

Figure A.18: Measurements for average data receive rate of client and orchestrating devices executing EM clustering algorithm

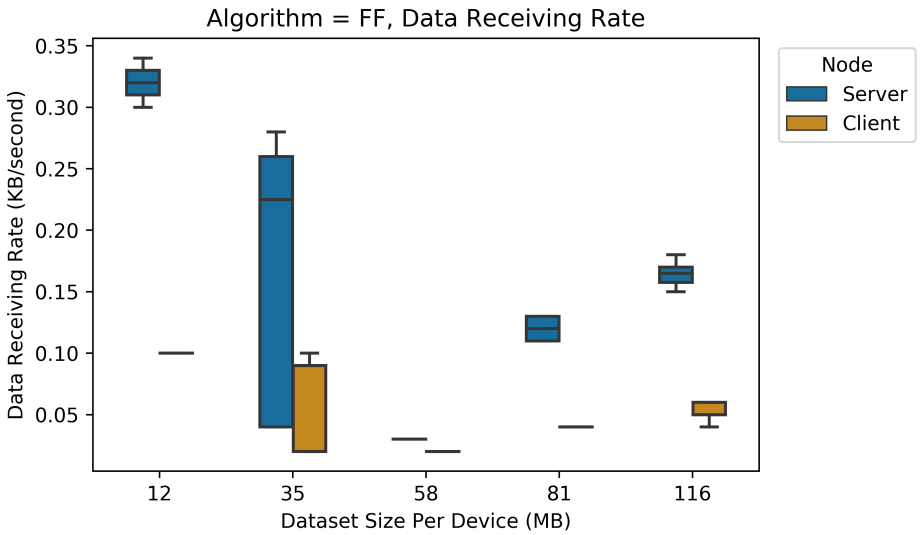

Figure A.19: Measurements for average data receive rate of client and orchestrating devices executing FF clustering algorithm

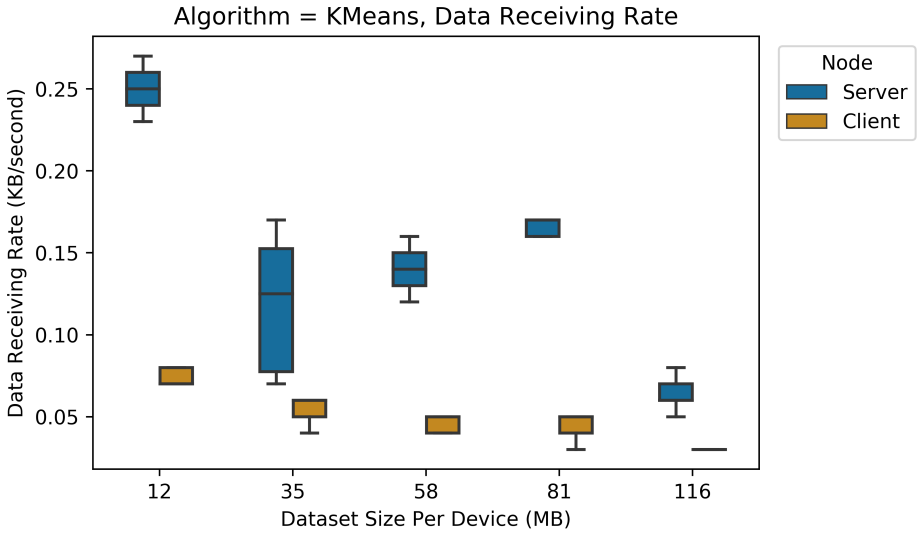

Figure A.20: Measurements for average data receive rate of client and orchestrating devices executing KMeans clustering algorithm

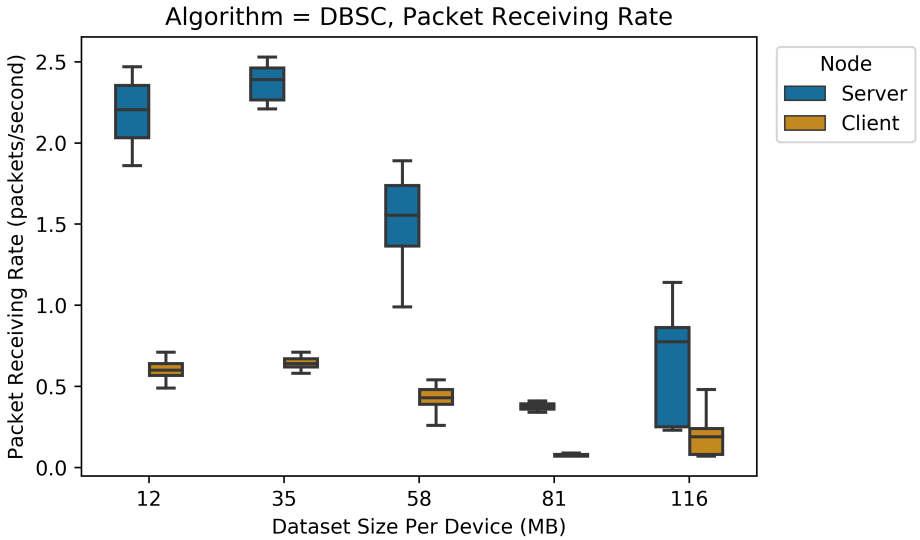

Figure A.21: Measurements for average packet receive rate of client and orchestrating devices executing DBSC clustering algorithm

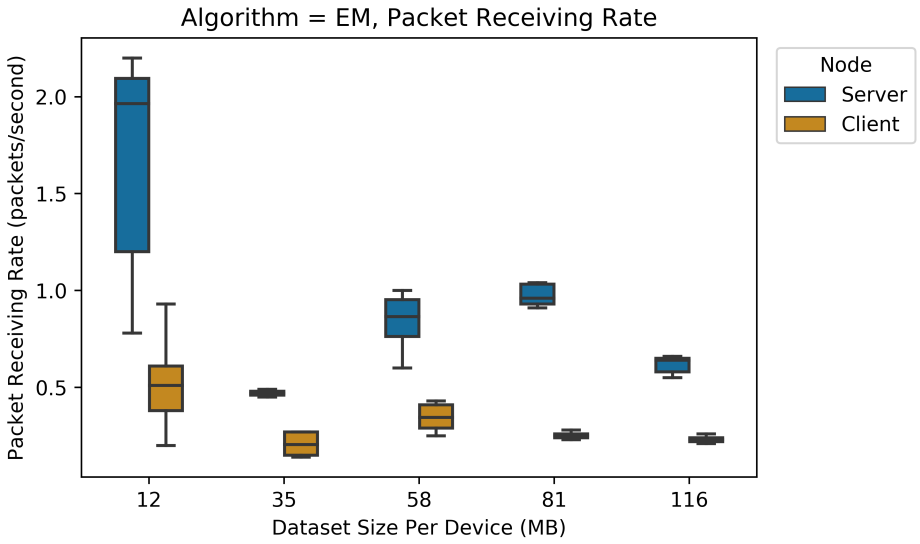

Figure A.22: Measurements for average packet receive rate of client and orchestrating devices executing EM clustering algorithm

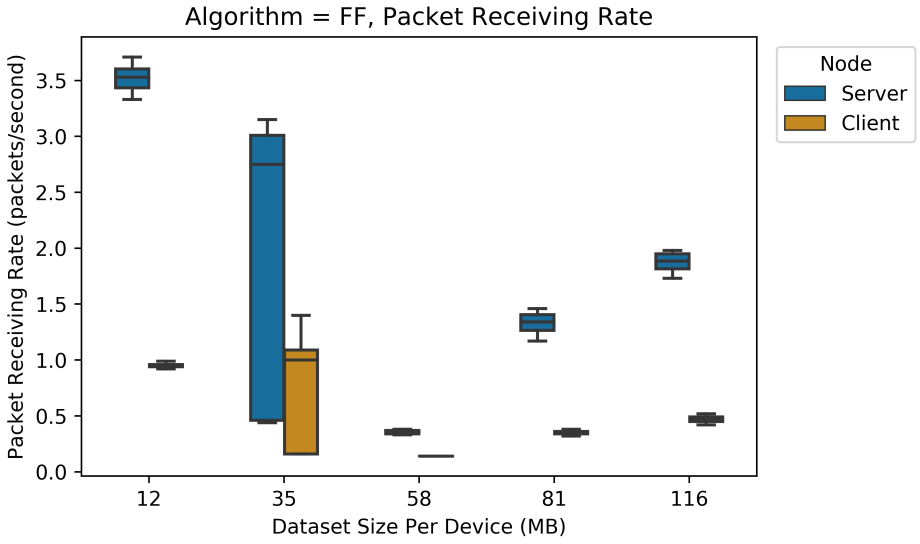

Figure A.23: Measurements for average packet receive rate of client and orchestrating devices executing FF clustering algorithm

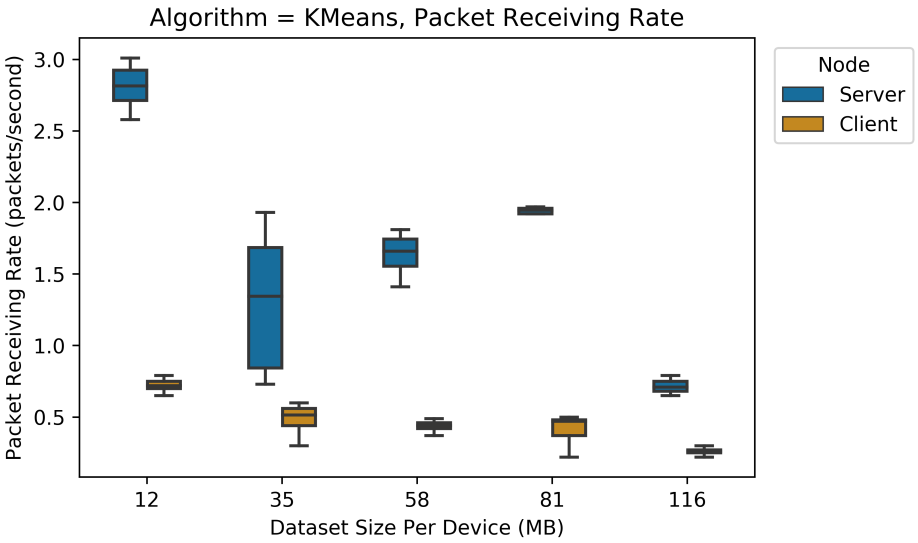

Figure A.24: Measurements for average packet receive rate of client and orchestrating devices executing KMeans clustering algorithm

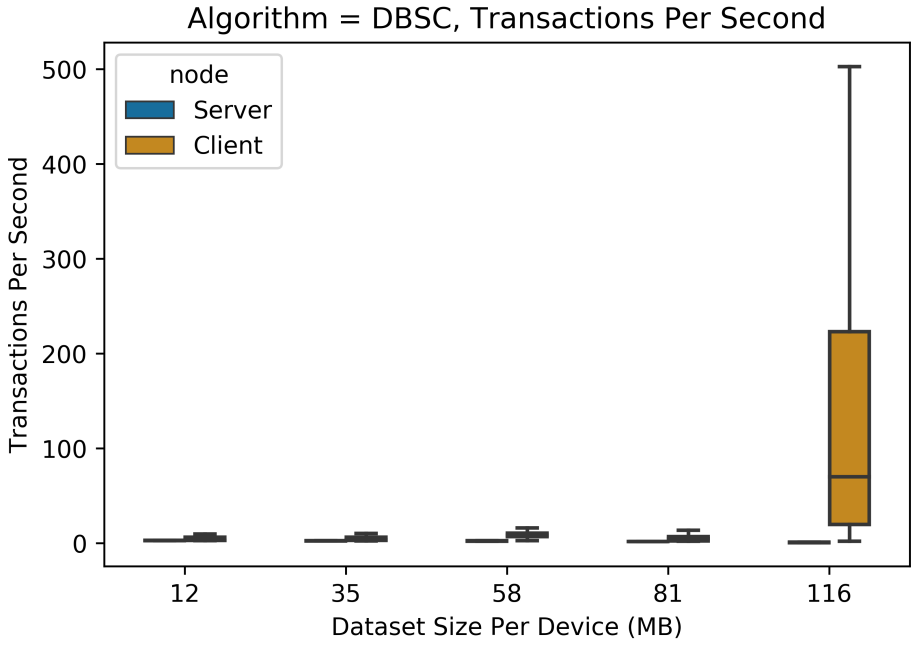

Figure A.25: Measurements for average read transactions per second of client and orchestrating devices executing DBSC clustering algorithm

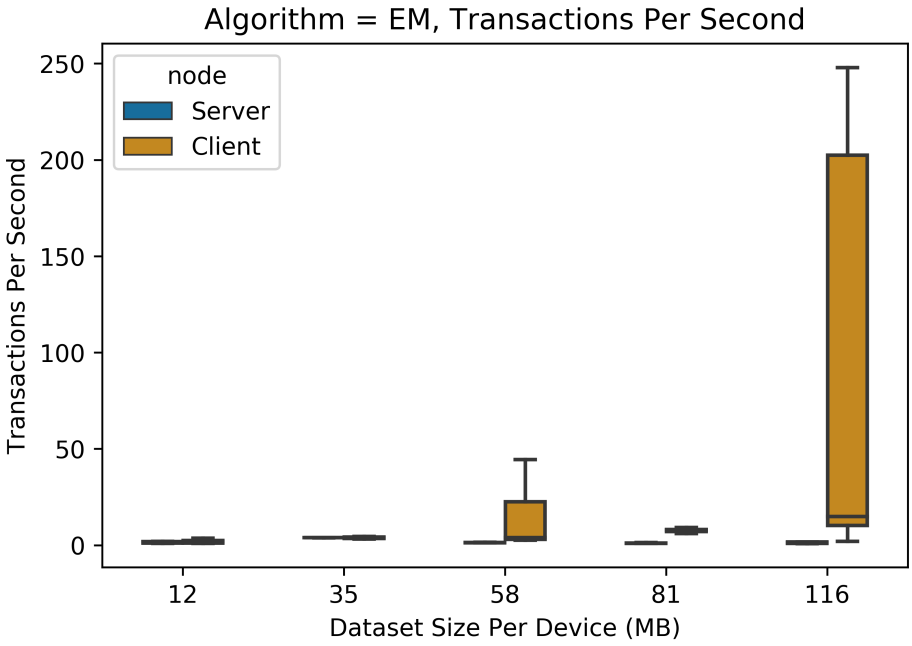

Figure A.26: Measurements for average read transactions per second of client and orchestrating devices executing EM clustering algorithm

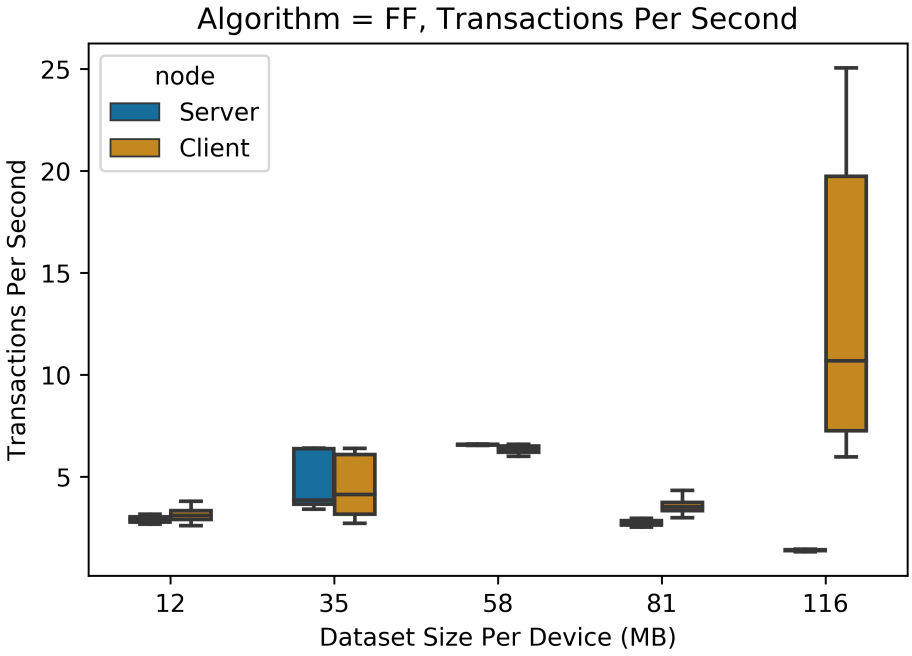

Figure A.27: Measurements for average read transactions per second of client and orchestrating devices executing FF clustering algorithm

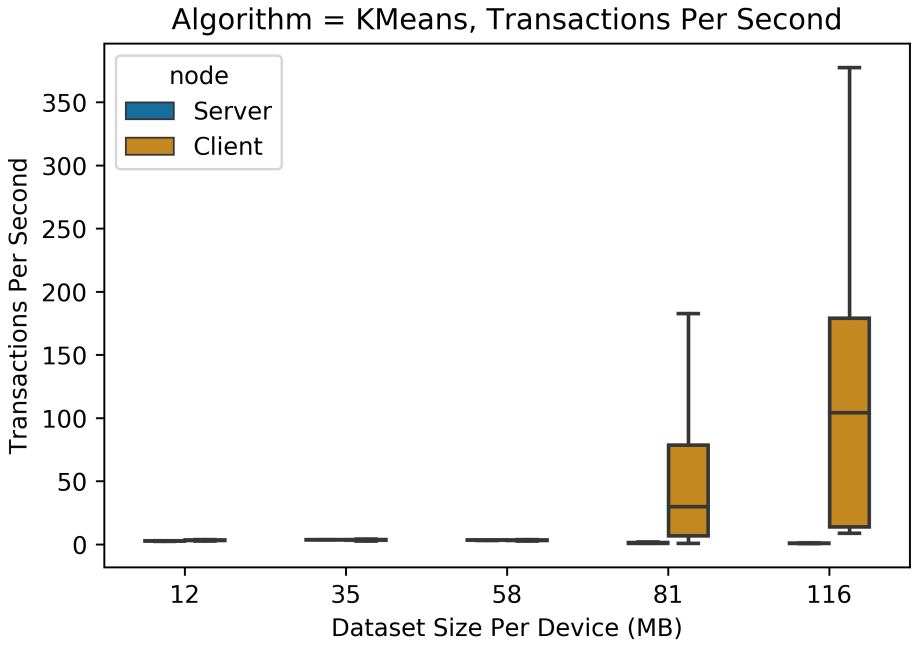

Figure A.28: Measurements for average read transactions per second of client and orchestrating devices executing KMeans clustering algorithm

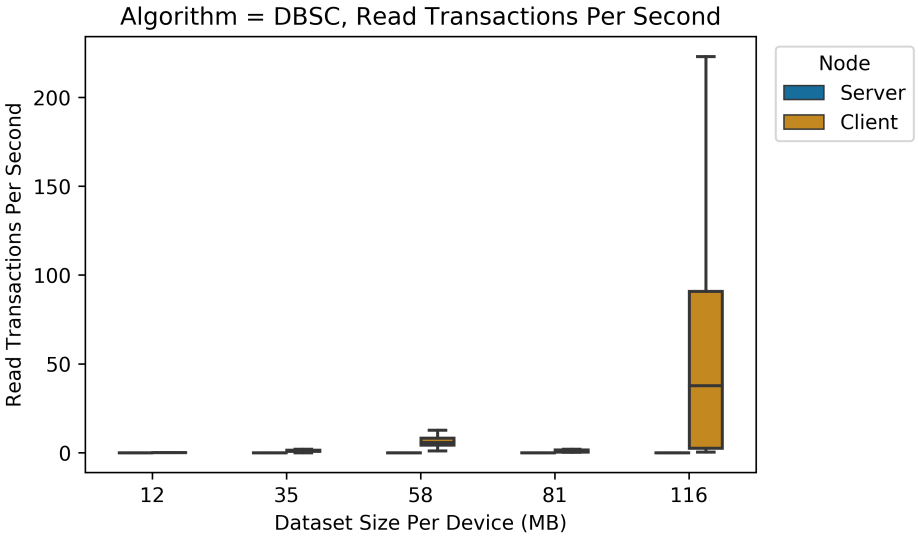

Figure A.29: Measurements for average read transactions per second of client and orchestrating devices executing DBSC clustering algorithm

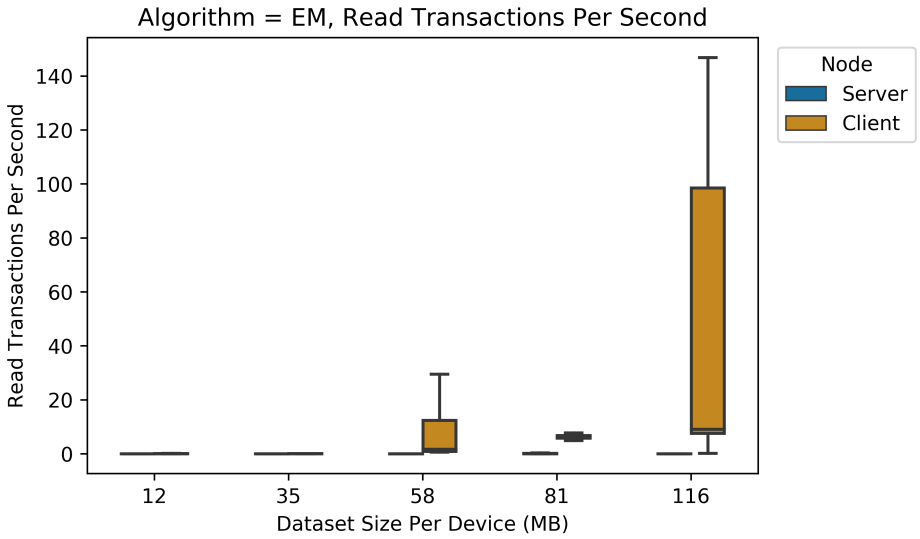

Figure A.30: Measurements for average read transactions per second of client and orchestrating devices executing EM clustering algorithm

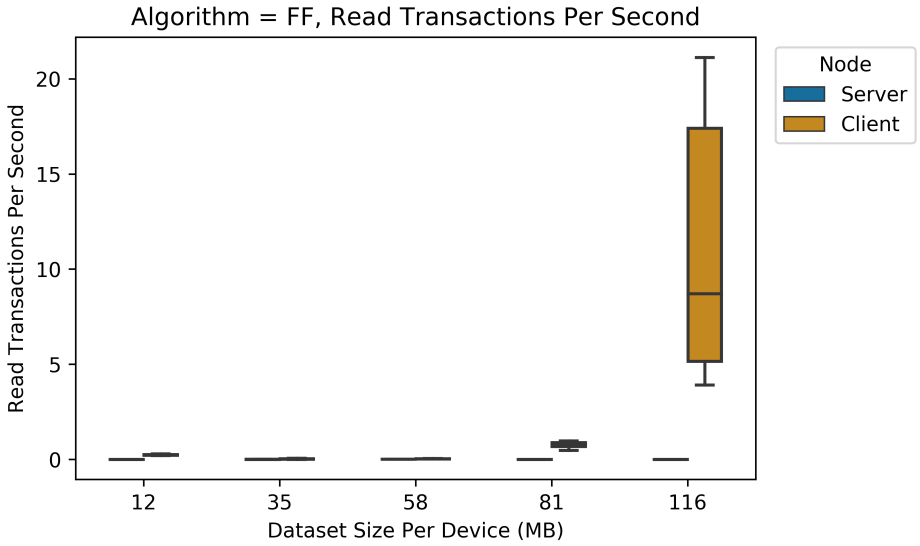

Figure A.31: Measurements for average read transactions per second of client and orchestrating devices executing FF clustering algorithm

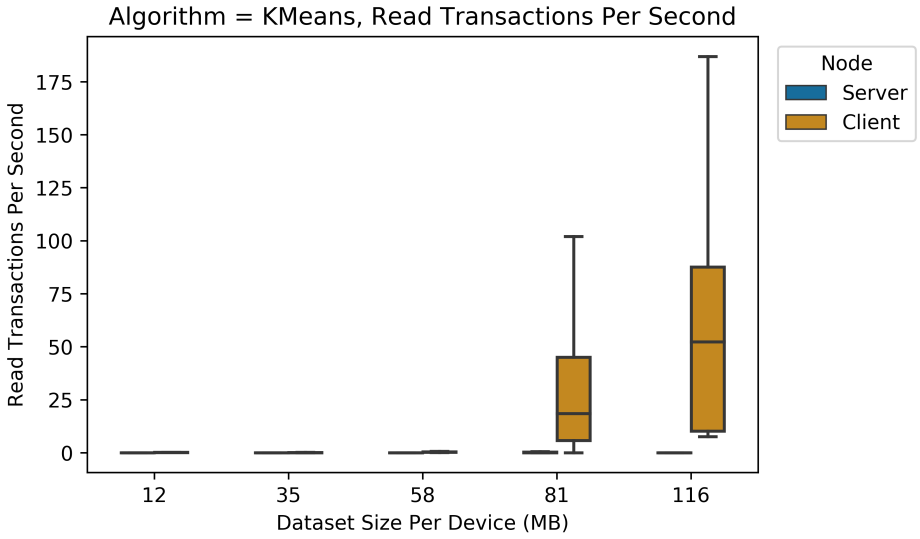

Figure A.32: Measurements for average read transactions per second of client and orchestrating devices executing KMeans clustering algorithm

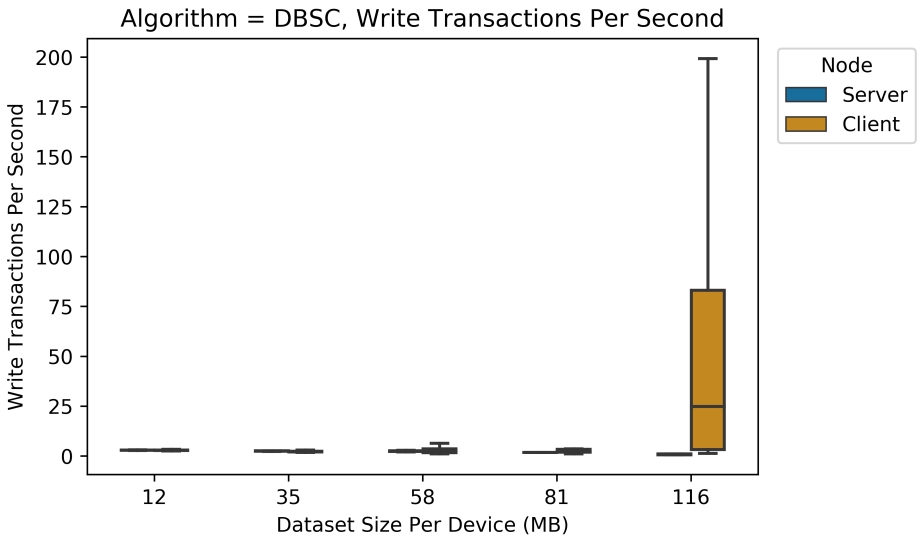

Figure A.33: Measurements for average write transactions per second of client and orchestrating devices executing DBSC clustering algorithm

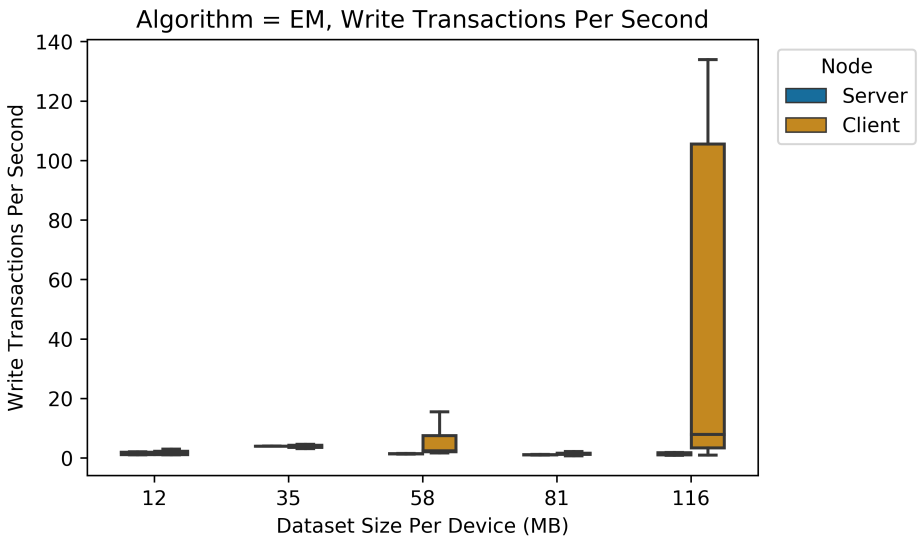

Figure A.34: Measurements for average write transactions per second of client and orchestrating devices executing EM clustering algorithm

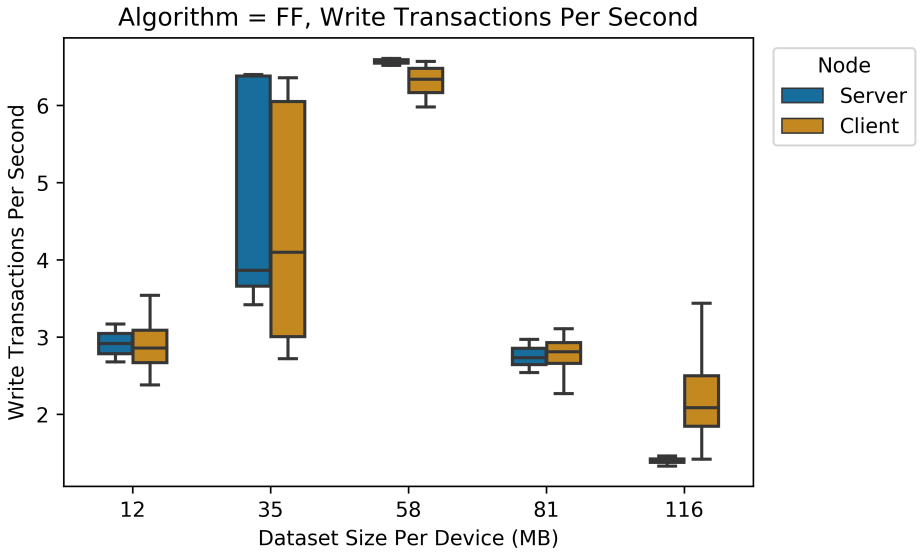

Figure A.35: Measurements for average write transactions per second of client and orchestrating devices executing FF clustering algorithm

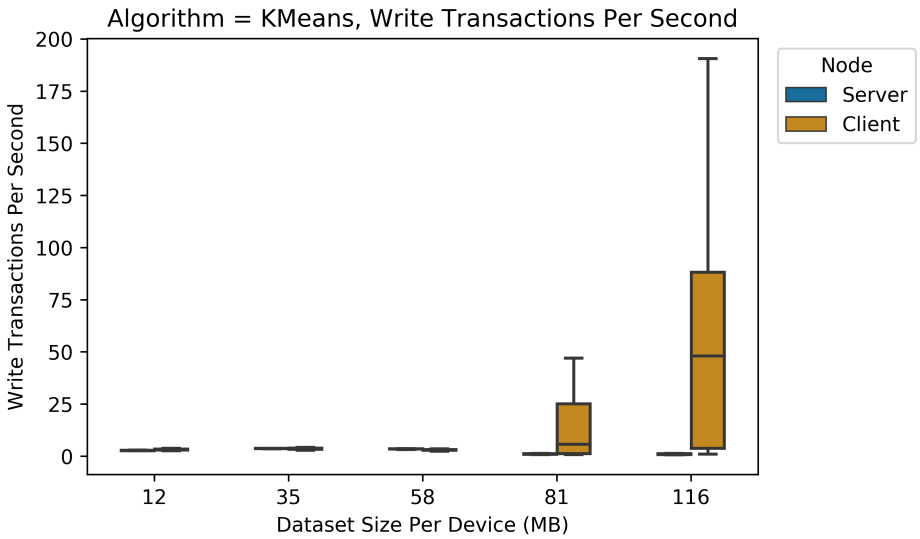

Figure A.36: Measurements for average write transactions per second of client and orchestrating devices executing KMeans clustering algorithm

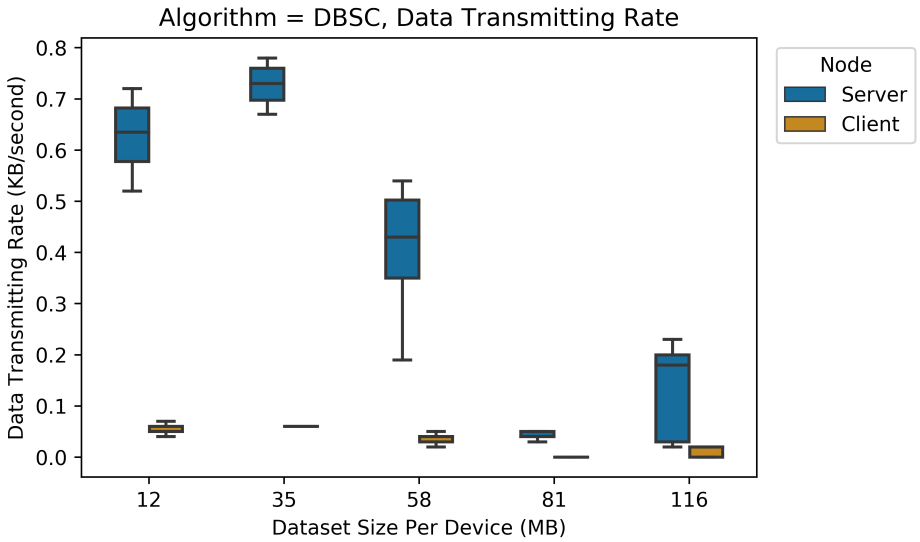

Figure A.37: Measurements for average data transmitting rate client and orchestrating devices executing DBSC clustering algorithm

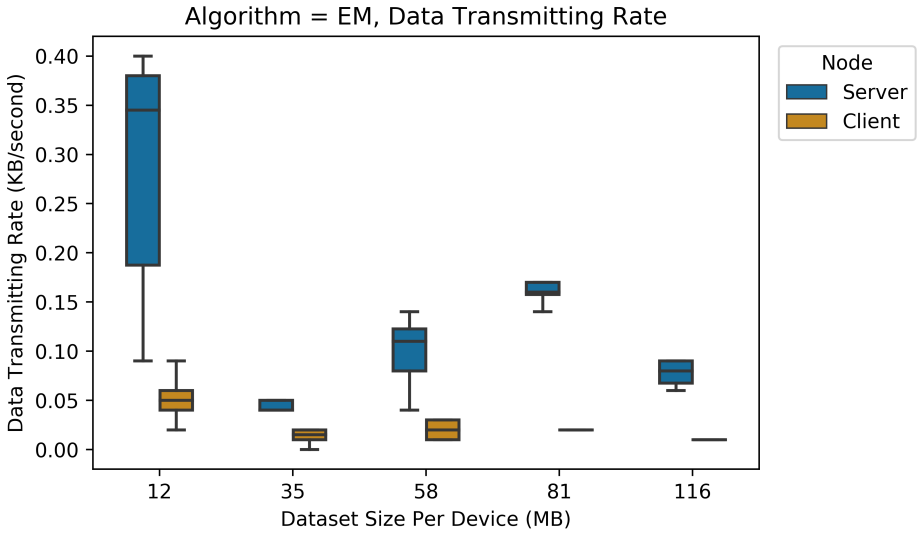

Figure A.38: Measurements for average data transmitting rate client and orchestrating devices executing EM clustering algorithm

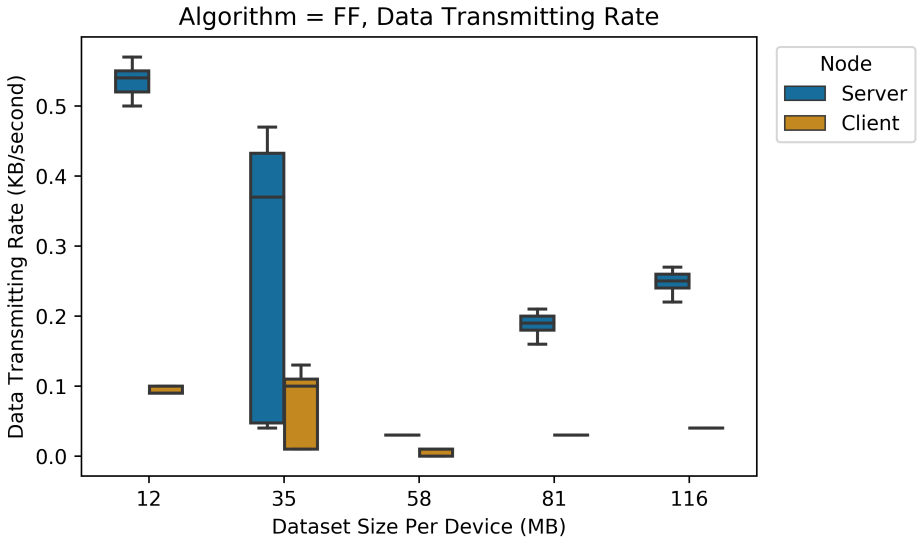

Figure A.39: Measurements for average data transmitting rate client and orchestrating devices executing FF clustering algorithm

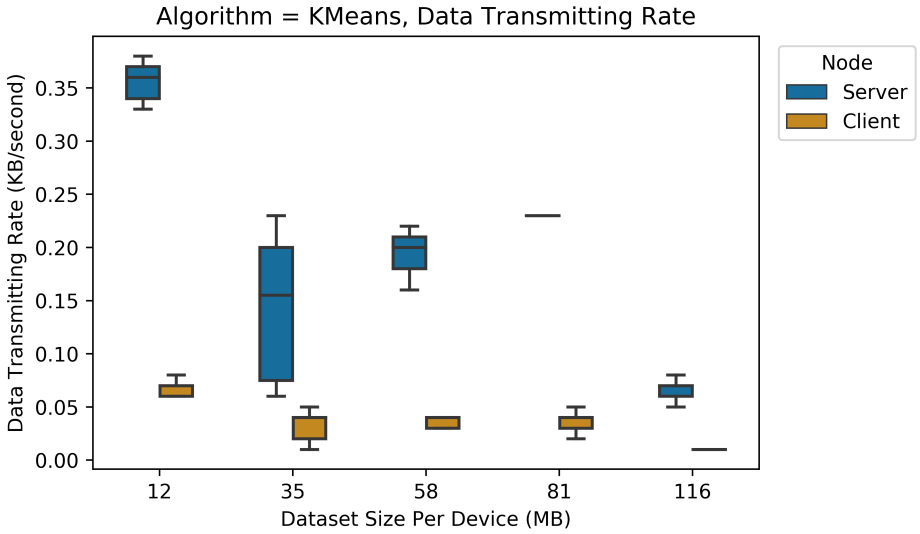

Figure A.40: Measurements for average data transmitting rate client and orchestrating devices executing KMeans clustering algorithm

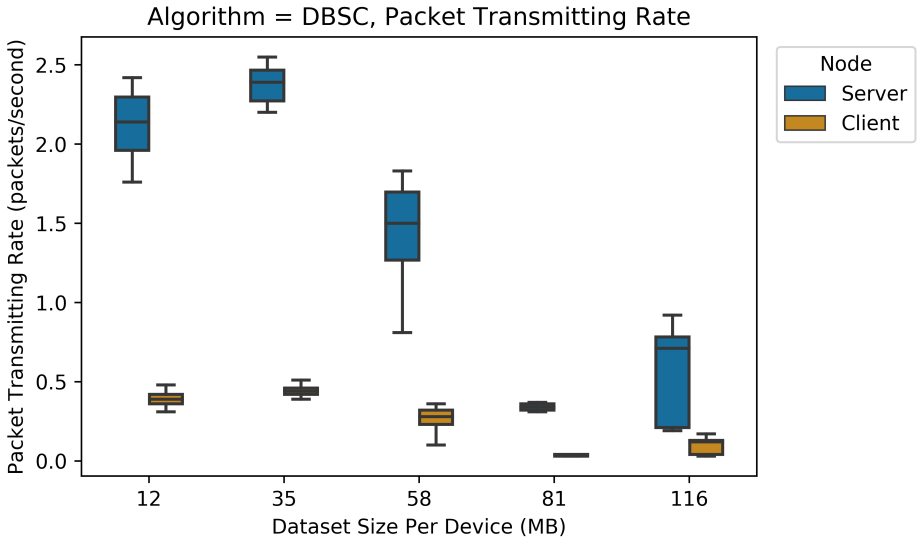

Figure A.41: Measurements for average packet transmitting rate client and orchestrating devices executing DBSC clustering algorithm

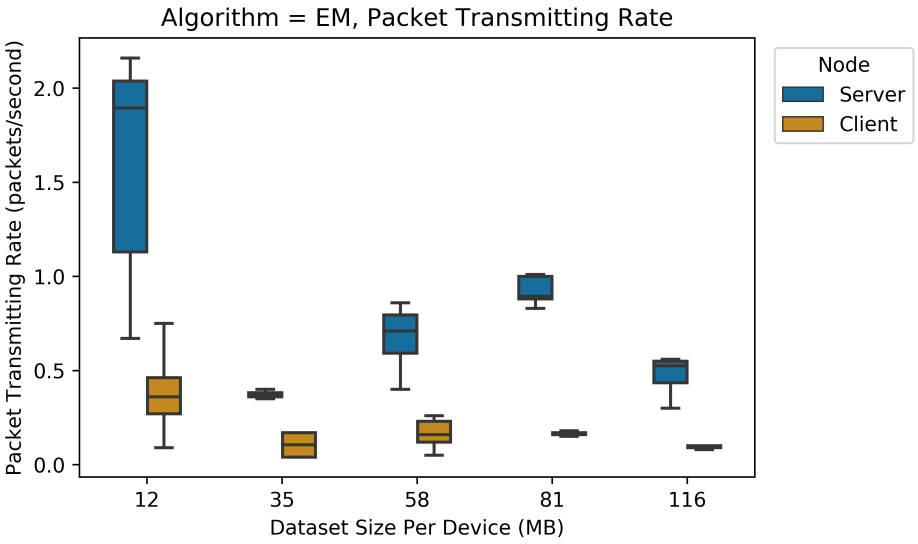

Figure A.42: Measurements for average packet transmitting rate client and orchestrating devices executing EM clustering algorithm

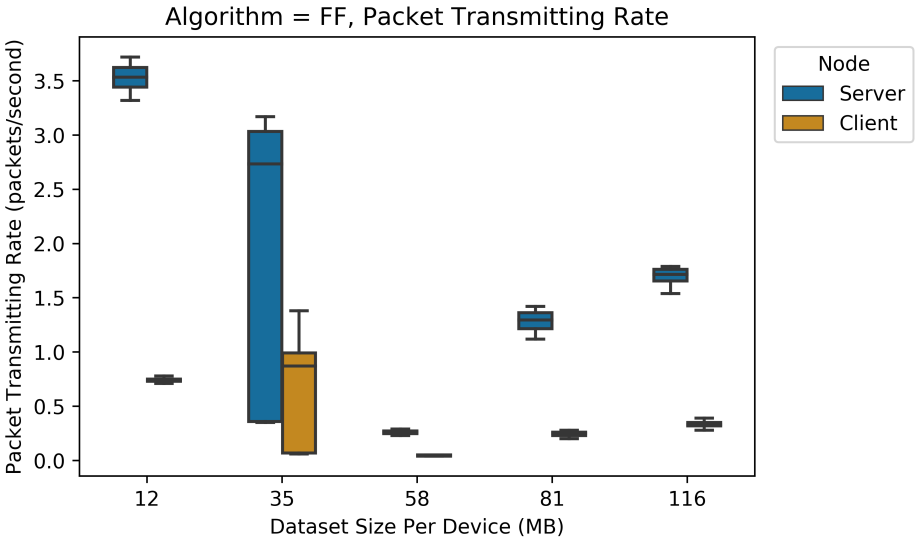

Figure A.43: Measurements for average packet transmitting rate client and orchestrating devices executing FF clustering algorithm

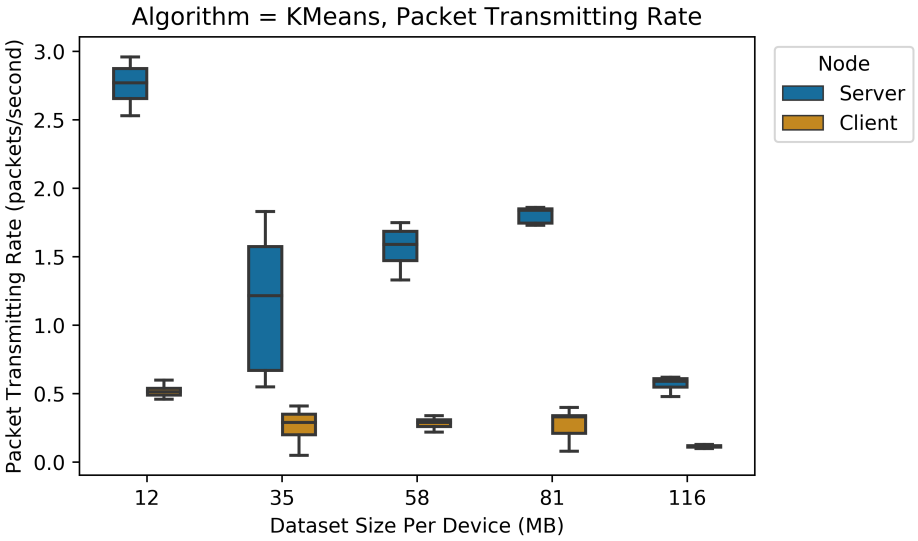

Figure A.44: Measurements for average packet transmitting rate client and orchestrating devices executing KMeans clustering algorithm

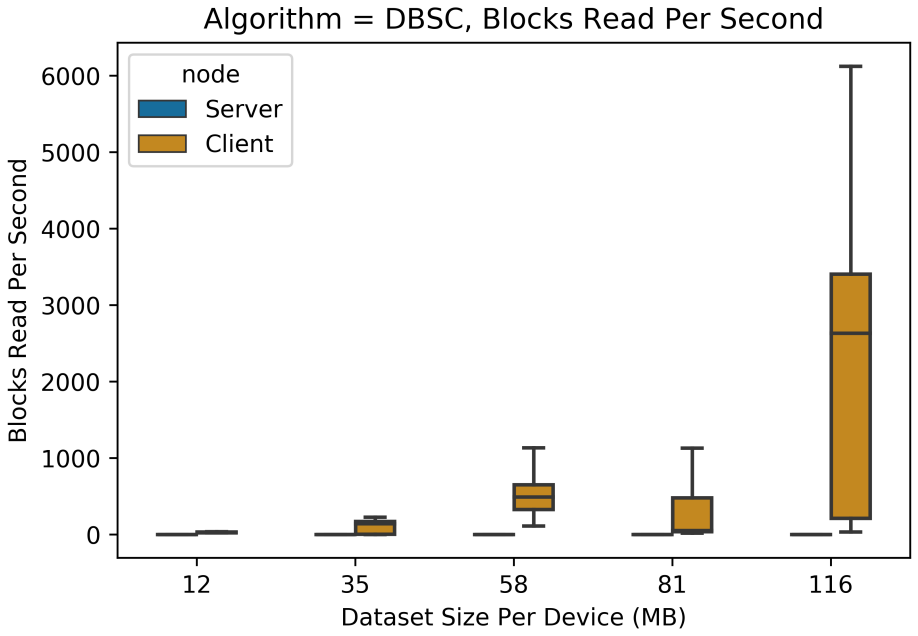

Figure A.45: Measurements for block reads per second for client and orchestrating devices executing DBSC clustering algorithm

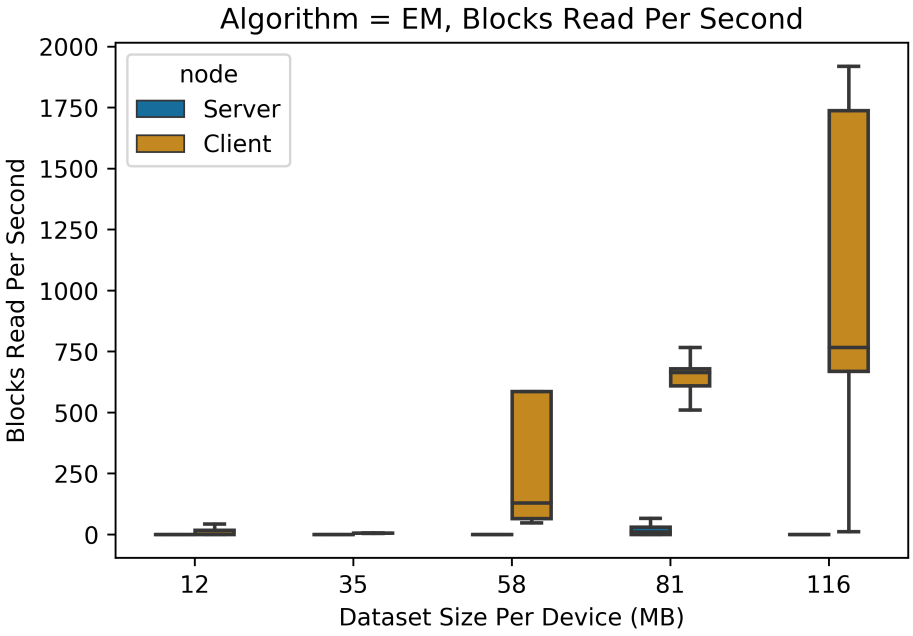

Figure A.46: Measurements for block reads per second for client and orchestrating devices executing EM clustering algorithm

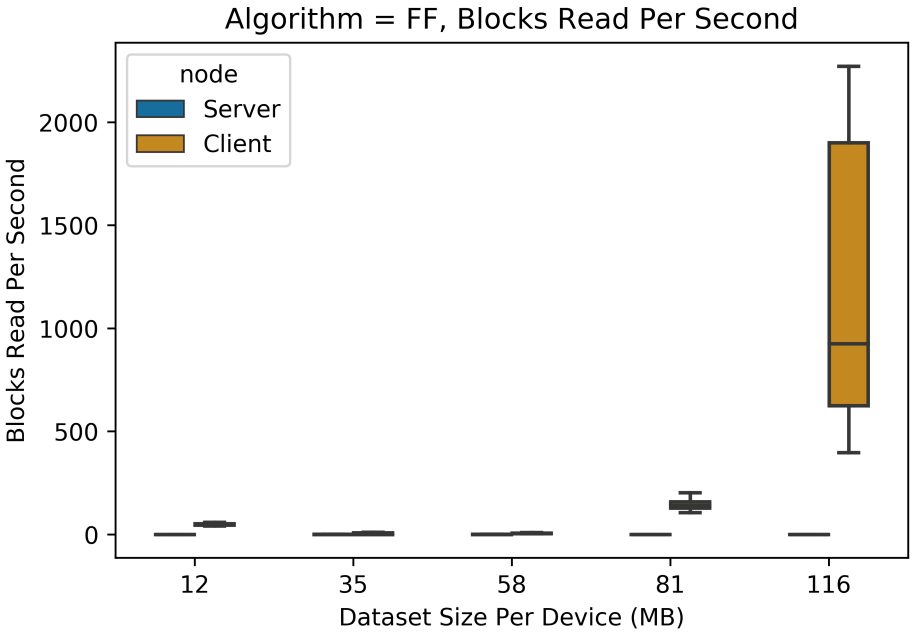

Figure A.47: Measurements for block reads per second for average packet transmitting rate client and orchestrating devices executing FF clustering algorithm

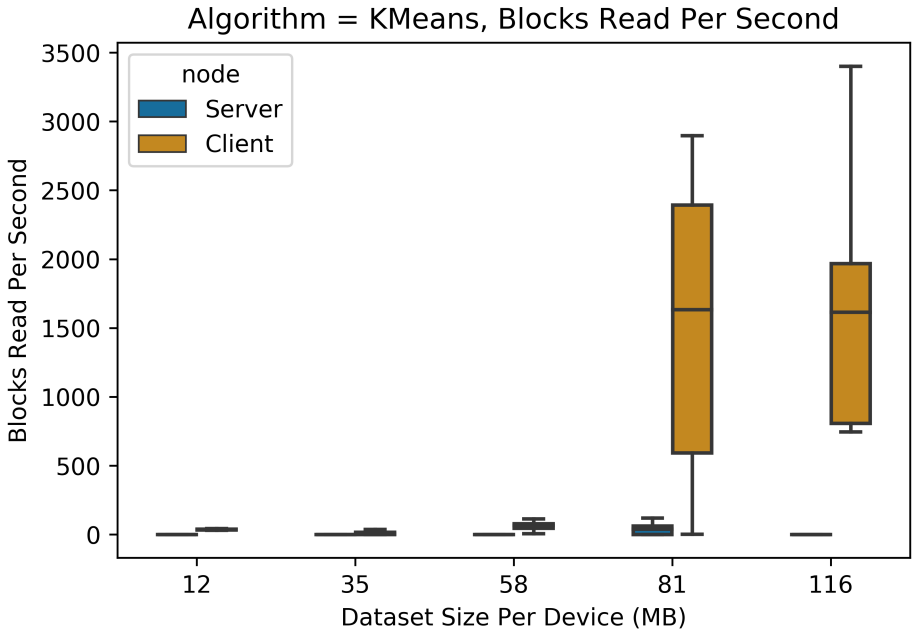

Figure A.48: Measurements for block reads per second for average packet transmitting rate client and orchestrating devices executing KMeans clustering algorithm

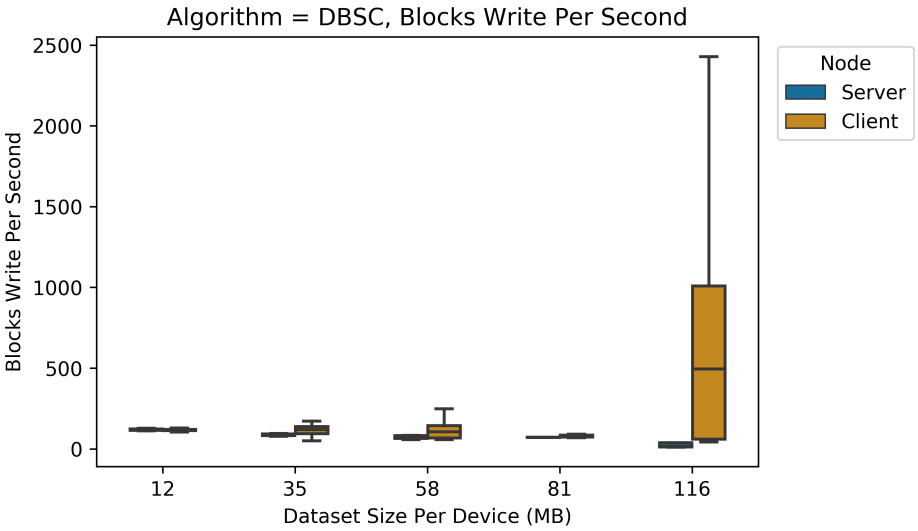

Figure A.49: Measurements for block write per second for client and orchestrating devices executing DBSC clustering algorithm

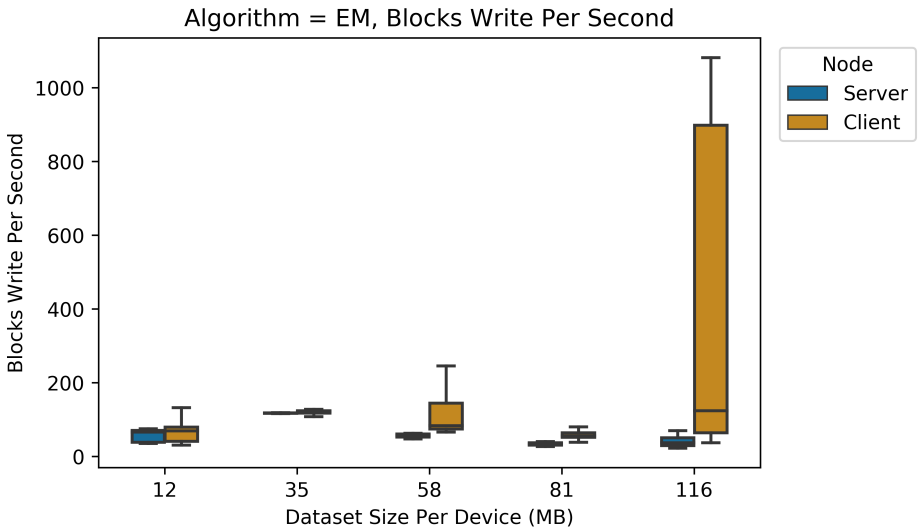

Figure A.50: Measurements for block write per second for client and orchestrating devices executing EM clustering algorithm

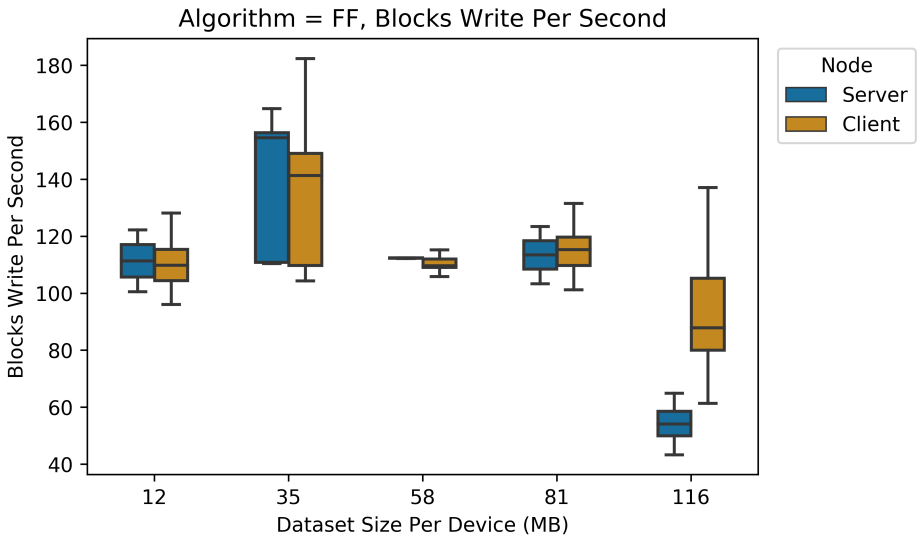

Figure A.51: Measurements for block write per second for average packet transmitting rate client and orchestrating devices executing FF clustering algorithm

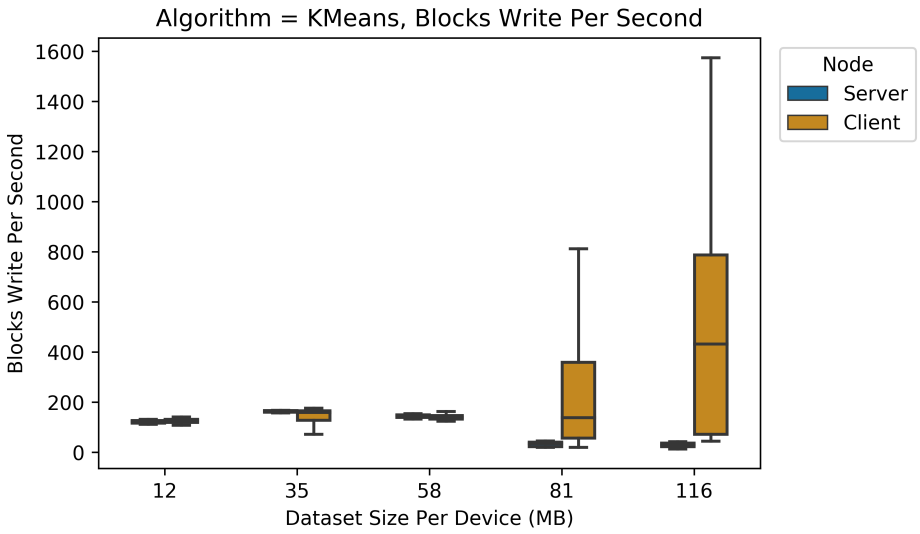

Figure A.52: Measurements for block write per second for average packet transmitting rate client and orchestrating devices executing KMeans clustering algorithm

Appendix B

Design of Possible User Studies

This appendix contains guidance for researchers interested in exploring the usability aspects of the technical contributions of this dissertation. Usability can be assessed by conducting user studies. Herein, we present some preliminary designs for such studies, including their research questions and rationale.

B.1 Usability of RICCi's Programming Model

RICCi is designed as an extension of Android's Inter-Component Communication (ICC) framework for distributed data access. RICCi's goals are to be an easy-to-learn, effective, and efficient middleware that helps developers access remote component data.

As an extension of ICC, RICCI follows the syntax that is very similar to the well-established ubiquitous ICC, implying that it should be easy-to-learn for Android developers, familiar with ICC. Also, in terms of effectiveness, RICCi is designed to handle the low-level details of the interactions between device-to-device communications, such as data-marshaling, and error handling, thus enabling developers to focus on higher-level details of their implementations. Lastly, with RICCi, developers have to think about how accessing the remote-component data will impact the network and runtime, implying that it is efficient as data transmission would not paper-over the network. In Chapter 3, we evaluated the performance of RICCi on custom-built applications. However, the aspects of usability w.r.t. learning-

	For each question, please circle the number that most appropriately reflects your impressions when using RICCi to implement a distributed mobile app:			
RQ1	1. I can use RICCi as easily as ICC to implement my application	disagree	0 1 2 3 4 5	agree
	2. I can use easily use RICCi to access remote data	disagree	0 1 2 3 4 5	agree
RQ2	3. I feel that the increased battery consumption is not an issue when deploying my edge-based application	disagree	0 1 2 3 4 5	agree
	4. I feel comfortable when using RICCi to access large remote data files	disagree	0 1 2 3 4 5	agree
RQ3	5. RICCi makes me think through how the data size will impact my application runtime	disagree	0 1 2 3 4 5	agree
	6. Latency when accessing remote data is my main concern when design distributed applications	disagree	0 1 2 3 4 5	agree
RQ4	7. RICCi's data exchange patterns help me decide how my application will behave at runtime	disagree	0 1 2 3 4 5	agree
	8. RICCi's data exchange patterns inform my design when implementing a distributed application	disagree	0 1 2 3 4 5	agree

Table B.1: Possible user study questions for developers using RICCi to implement distributed applications

curve, effectiveness, and efficiency remain unexplored.

We outlined the following possible research questions as guidance to researchers interested in investigating the usability aspects of RICCi. Please keep in mind that these questions are not restrictive and only serve as possible first steps towards building upon our work. **RQ1:** Can developers familiar with ICC use RICCi to implement distributed applications with minimal training? **RQ2:** Do RICCi's data-transmission methodologies facilitate remote data-access for experienced Android developers? **RQ3:** Do RICCi developers understand the impact of remote data transmission on app performance? **RQ4:** Do RICCi's data-transmission programming interfaces help developers make optimal design choices?

The purpose of RQ1 is to determine how easy RICCi is to learn. The purpose of RQ2 is to determine if developers can use RICCi to access remote data. The purpose of RQ3 is to determine how developers perceive the impact of their choices when transmitting data across the network. The purpose of RQ4 is to determine if RICCi's data-transmission patterns effectively guide developers' decision-making process.

The theme of RQ1 and RQ2 is to determine how quickly developers can apply the RICCi

framework to solve their design problems. The theme of RQ3 and RQ4 is to determine if the advantages of RICCi are perceived positively by the developers. We suggest that researchers interested in answering these motivating questions conduct a user study targeting Android developers, who have had prior experience with distributed applications. In this study, developers should be placed into two groups. Both groups should receive the same assignment: implement an Android app that retrieves files stored at a remote Android device. Each data object must have a different set of properties. For example, a small text file, a music file, and an uncompressed video file. The groups should receive the RICCi framework and documentation. However, one group should receive additional training. Both groups should implement the task twice, once without RICCi and another using it. Having implemented their apps, participants would then be asked to complete questionnaires detailing their experiences. The questionnaire would contain a Likert scale with a range of values describing how likely the participants are to agree with a statement. Table B.1 shows a sample questionnaire. With the results at hand, the results should be able to help researchers determine if RICCi is easy-to-learn and effective when aiding developers with the implementation of distributed applications. Also, the study will inform researchers how efficient and effective RICCi is in helping developers implement their distributed mobile applications.

B.2 Usability of Edge-based Architecture for Participatory Sensing

As previously discussed in Chapter 4, the main result of our work on creating exemplary use cases for the development of edge-based applications is our architecture for edge-based participatory sensing. It targets scenarios requiring easy access to computing resources, low network bandwidth, and having several co-located devices (e.g., shopping malls, train sta-

tions, cruise ships, etc.). In our reference implementation, the runtime automatically handles device coordination and rewards distribution via a workflow script. In terms of efficiency, our measurements show that our reference implementation requires low latency between the edge and the cloud, with additional energy requirements 4.3. Without our architecture, system developers would have to optimize their system to minimize the transmitted data volume. Thus incurring additional effort and time, this would likely introduce bugs and security vulnerabilities to the system. However, the aspects of usability for our contributions in terms of being practical and effective remain unexplored.

The two main stakeholders of this contribution are edge-device *users* and edge-application *developers*, whose requirements differ. Users value quality-of-service, low-energy consumption, quick response time, and high rewards. Developers value low-effort and high expressiveness when implementing their applications.

We outline the following questions to motivate user studies to determine if our contributions are practical and effective to users and developers. **RQ1:** Can developers express their design ideas by using job scripts to define edge-based tasks? **RQ2:** Are developers concerned with the host device's increased battery consumption and the reduced network utilization? **RQ3:** How users perceive the tradeoff of increased energy consumption and reduced network utilization? **RQ4:** How likely are incentive rewards to motivate users to participate? **RQ5:** Are incentive rewards effective to ensure that users provide truthful data?

The purpose of RQ1 is to determine the effectiveness of job scripts for defining edge-based tasks. RQ2 and RQ3 aim to determine how users and developers perceive the performance tradeoffs of the edge architecture. The purpose of RQ4 and RQ5 is to determine if incentive reward distribution is effective for ensuring user participation, as well as to motivate users to contribute authentic data to the system.

	For each question, please circle the number that most appropriately reflects your impressions about developing the edge-based application:			
RQ1	1. I can use the IDE to effectively define my application workflow	disagree	0 1 2 3 4 5	agree
	2. I feel that the JSON workflow script helped me implement my edge app	disagree	0 1 2 3 4 5	agree
RQ2	3. I feel that the increased battery consumption is not an issue when deploying my edge-based application	disagree	0 1 2 3 4 5	agree

Table B.2: Possible user study questions targeting edge-application developers

	For each question, please circle the number that most appropriately reflects your impressions while using the application:			
RQ3	1. I feel that the app used too much battery	disagree	0 1 2 3 4 5	agree
	2. I am concerned with how much data the device transmits over the network	disagree	0 1 2 3 4 5	agree
RQ4	3. I feel that the increased battery consumption is not an issue when deploying my edge-based application	disagree	0 1 2 3 4 5	agree
	4. I feel that the incentive rewards are well balanced in terms of how much I contributed to the app	disagree	0 1 2 3 4 5	agree
RQ5	5. I feel motivated to provide accurate data to the app	disagree	0 1 2 3 4 5	agree

Table B.3: Possible user study questions targeting device users

We suggest that researchers interested in answering those motivating questions conduct user studies targeting edge-based service marketplaces. To that end, researchers may investigate different approaches to facilitate the creation of the job scripts (e.g., tool, plugins). Researchers may also investigate if developers perceive the performance tradeoff (i.e., increase energy consumption and reduced latency) as practical for their application under design. Also, researchers may choose to investigate the perceived users' usability of an edge application that uses our architecture. Table B.2 and Table B.3 show questionnaires that researchers may use to gather feedback from developers and users, respectively.

B.3 Usability of Stargazer

Stargazer is designed as a command-line tool to inform developers about the performance and resource-utilization metrics of their applications under design. In Chapter 6, we provide the details of our methodology for performance estimation and how practitioners can use

For each question, please circle the number that most appropriately reflects your experience while using the tool for developing your application				
RQ1	1. The tool was able to help me reduce the energy consumption of my app	disagree	0 1 2 3 4 5	agree
	2. Tool informs me about how my design choices affect the CPU utilization	disagree	0 1 2 3 4 5	agree
RQ2	3. Tool's output made me fundamentally improve my original design	disagree	0 1 2 3 4 5	agree
	4. I trust the tool's output when making changes to my original code to improve its performance	disagree	0 1 2 3 4 5	agree
RQ3	5. The tool effectively displayed the information necessary to improve my app's performance	disagree	0 1 2 3 4 5	agree
	6. I understand the output from the tool and how it is correlated to my design	disagree	0 1 2 3 4 5	agree

Table B.4: Possible user study questions for researcher investigating how effective is a performance estimation tool when informing developers about their design choices

it to guide their design decisions. Section 6.3 discusses how to apply our methodology to a specific use case. Nevertheless, the usability aspect of Stargazer remains unexplored. In this section, we present suggestions for research directions that practitioners may follow to evaluate the usability of a Stargazer-type tool in terms of its effectiveness. A starting point for a usability study for a Stargazer-type tool could be to understand how to best inform developers about how their code impacts the system performance. To that end, we outline the following possible research questions to guide such studies. **RQ1**: How effective is the tool when informing developers about their design choices? **RQ2**: Did the feedback from the tool fundamentally change the developer's design? **RQ3**: How to best display the information to developers?

The purpose of RQ1 and 2 are to determine if the tool is effective when informing developers about their design decisions. The purpose of MQ3 is to determine what is the most effective way to display the tool's output information to the developers.

We suggest that researchers interested in answering these motivating questions conduct a user study targeting edge application developers. In this study, developers should be tasked

with implementing an edge-based application that meets a set of performance requirements (e.g., memory utilization, energy consumption, task duration). Participants are required to interact with the tool by using an IDE or a CLI and use its information to guide their design process. Upon completing the task, the participants should complete a questionnaire to provide their feedback about the tool's effectiveness. Table B.4 shows an example of questionnaire that researchers may use to gather feedback.

Appendix C

User Manual: Performance Estimation Tool

This appendix contains the steps to execute the code of our empirical study and to use STARGAZER as presented in Chapter 5.

C.1 Empirical Study

Please follow the steps below to run the code of the empirical study (Section 5.2).

1. Download the availability package from https://github.com/brenodan/stargazer.
2. Unzip the *startgazer-master.zip* file.
3. Open the "empirical_study" folder.
    - The scripts folder contains the scripts used to manage the client devices.
    - Each script file follows a naming convention. The first number refers to the device id and the second to the experiment run (e.g., "driver-0-1.sh" – device 0 & first run).
    - src_code folder contains the source code of applications used during the data collection.

4. Place the scripts for devices from 0 to 7 in clients devices. Also, place the scripts for device 8 in the orchestrator device.

5. Place the source code of the client and end-point files in the client devices.

6. Place the source code of the server and driver in the server device.

7. Update the Utils.java in the server device with the IP addresses of the clients.

8. Run the end-point programs in each client device.

9. Recompile the server program with the updated IP table.

10. Start the driver program on the server device.

11. Upon completing the tasks, each client places its performance metrics in the Dataset directory in the Desktop (user/Desktop/Dataset).

C.2 Running Stargazer

Stargazer is tool that can be used to estimate the performance and resource utilization of an edge-based system. Below we outline the steps to estimate the average *energy consumption* of your system. You may use the other sample Python scripts in the repository to estimate other metrics such as CPU utilization, task duration, etc.

1. Download the availability package from https://github.com/brenodan/stargazer.

2. Unzip the *startgazer-master.zip* file.

3. Unzip the *dnn source code and graphs.zip* file.

4. Open the "dnn source code and graphs" folder.

5. Place the training data in the same directory of the $\mathrm{ml}_{s}ystem_{p}rediction_{e}nergy.pyfile.Placeatestfileu$

6. Update the $\mathrm{ml}_{s}ystem_{p}rediction_{e}nergy.pywiththepathtothetestfile.Runthepythoncode(\$python3ml_{s}ystem$

7. Upon finishing, the energy prediction results will be printed out on the terminal.