5-2018

Universal Mobile Service Execution Framework for Device-To-Device Collaborations

Minh Le
Utah State University

UNIVERSAL MOBILE SERVICE EXECUTION FRAMEWORK FOR DEVICE-TO-DEVICE COLLABORATIONS

by

Minh Le

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

Stephen W. Clyde, Ph.D. Major Professor	Vicki H. Allan, Ph.D. Committee Member
Curtis Dyreson, Ph.D. Committee Member	Haitao Wang, Ph.D. Committee Member
Qingyang Hu, Ph.D. Committee Member	Mark R. McLellan, Ph.D. Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2018

ABSTRACT

UNIVERSAL MOBILE SERVICE EXECUTION FRAMEWORK FOR DEVICE-TO-DEVICE COLLABORATIONS

by

Minh Le, Doctor of Philosophy

Utah State University, 2018

Major Professor: Stephen W. Clyde, Ph.D.
Department: Computer Science

There are high demands of effective and high-performance of collaborations between mobile devices in the places where traditional Internet connections are unavailable, unreliable, or significantly overburdened, such as on a battlefield, disaster zones, isolated rural areas, or crowded public venues. One solution is to form opportunistic networks and distribute certain computations to peer devices with unused computational capacity. To enable collaboration among the devices in opportunistic networks, code offloading and Remote Method Invocation are the two major mechanisms to ensure code portions of applications are successfully transmitted to and executed on the remote platforms. Although these domains are highly enjoyed in research for a decade, the limitations of multi-device connectivity [1, 2], system error handling [3] or cross platform compatibility [4] prohibit these technologies from being broadly applied in the mobile industry.

This dissertation addresses five critical technical problems that are blocking the way of applying device-to-device communication to improve performance and preserve energy in mobile industry: (1) lack of optimized algorithm for task division to efficiently split and distribute tasks fairly to all nearby ones base on their resource availability at the moment of request making, (2) integration of existing middleware architecture to edge platforms to detect system and network malfunctions, then backup and yield in-progress tasks to the nearby mobile networks, (3) missing an universal code offloading infrastructure for task to distribute and get resolved over heterogeneous mobile platform, (4) the essential limitations of collaboration on mobile devices including: *short distance connections*: 200 meters for WiFi-Direct or 10 meters for Bluetooth, *high cost of service development*: developer takes a tremendous amount of time to develop an execution package following the middleware template, *slow response*: inapplicable for real-time applications, finally (5) use of method invocation for effective code distribution and offloading over multi-group device-to-device networks.

To address the above problems, we designed and developed UMSEF – an Universal Mobile Service Execution Framework, which is an innovative and radical approach for mobile computing in opportunistic networks. Particularly, our solution comprises of the following contributions: (1) a component-based mobile middleware architecture that is flexible and adaptive with multiple network topologies, tolerant for network errors and compatible for multiple platforms, (2) an effective algorithm to estimate the resource availability of a device for higher performance and energy consumption, and (3) a novel platform for mobile remote method invocation based on declarative annotations over multi-group device networks. The experiments in reality exposes our approach not only achieve the better performance and energy consumption, but can be extended to large-scaled ubiquitous or IoT systems.

This dissertation contains five conference papers, of which four are already published in conference proceedings (SAC 2017 [5], FMEC 2017 [6], COMPSAC 2017 [7], iiWAS 2017 [8]) and one that has been submitted to MOBILESoft 2018 and is still under review.

(168 pages)

PUBLIC ABSTRACT

UNIVERSAL MOBILE SERVICE EXECUTION FRAMEWORK FOR DEVICE-TO-DEVICE COLLABORATIONS

Minh Le

There are high demands of effective and high-performance of collaborations between mobile devices in the places where traditional Internet connections are unavailable, unreliable, or significantly overburdened, such as on a battlefield, disaster zones, isolated rural areas, or crowded public venues. To enable collaboration among the devices in opportunistic networks, code offloading and Remote Method Invocation are the two major mechanisms to ensure code portions of applications are successfully transmitted to and executed on the remote platforms. Although these domains are highly enjoyed in research for a decade, the limitations of multi-device connectivity, system error handling or cross platform compatibility prohibit these technologies from being broadly applied in the mobile industry.

To address the above problems, we designed and developed UMSEF – an Universal Mobile Service Execution Framework, which is an innovative and radical approach for mobile computing in opportunistic networks. Our solution is built as a component-based mobile middleware architecture that is flexible and adaptive with multiple network topologies, tolerant for network errors and compatible for multiple platforms. We provided an effective algorithm to estimate the resource availability of a device for higher performance and energy consumption and a novel platform for mobile remote method invocation based on declarative annotations over multi-group device networks. The experiments in reality exposes our approach not only achieve the better performance and energy consumption, but can be extended to large-scaled ubiquitous or IoT systems.

To my parents, wife and son

ACKNOWLEDGMENTS

The PhD program is such a wonderful journey where I could explore the new research in my area of interest and discover my new limitation in the ocean of knowledge. And, this journey would not be accomplished without help from many individuals.

First, I would like to express my deepest appreciation to my advisor, Dr. Stephen W. Clyde, for his immeasurable support and guidance for my graduate program and future career. His invaluable instructions have helped me extend my research from the small scope to the final achievements including conference papers and dissertation. Besides, Dr. Clyde has also been a great advisor for my future career, without his support, I would not be able to reach to this point so easily.

I would also sincerely thank Dr. Young-Woo Kwon for his priceless advice and instructions for the first two years of my program. He was my first advisor who brought me to US and helped me change my thoughts from a software developer to a researcher. He also helped me build the base for my future research.

I would like to especially thank my committee members, Dr. Vicki H. Allan, Dr. Curtis Dyreson, Dr. Haitao Wang and Dr. Qingyang Hu, for carefully reading my dissertation and contributing valuable feedbacks that helped me improve the overall quality of my dissertation. I would also like to thank Genie Hanson, Vicki Anderson and Cora Price for their advice and help that ease any troubles during the time I work at the CS department.

My PhD journey would have not been possible without my family. I would like to thank my parents for their love, help and encouragement. Specially, I would like to thank my wife and my lovely son for their patient and sacrifice. They are my motivation on this journey and always by my side for every moment.

Finally, I am grateful to all advisors and friends at the CS and ECE departments at Utah State University. Their help, support and friendship are really helpful during my PhD program.

Minh Le

CONTENTS

Page

ABSTRACT . . . iii

PUBLIC ABSTRACT . . . v

ACKNOWLEDGMENTS . . . vii

LIST OF TABLES . . . xi

LIST OF FIGURES . . . xii

LIST OF CODE SNIPPETS . . . xv

ACRONYMS . . . xvi

1 INTRODUCTION . . . 1
- 1.1 Task Partitioning And Distribution . . . 3
- 1.2 Highly Dynamic And Volatile Edge Systems . . . 4
- 1.3 Offloading Of General-Format Packages . . . 7
- 1.4 Flexible N-N Model For Device-To-Device Networks . . . 9
- 1.5 Enabling RMI On Mobile Platforms . . . 10
- 1.6 Offloading In Multi-Group In Mobile Applications . . . 12
- 1.7 Summary . . . 12

2 UTILIZING NEARBY COMPUTING RESOURCES FOR RESOURCE-LIMITED MOBILE DEVICES . . . 14
- 2.1 Abstract . . . 14
- 2.2 Introduction . . . 14
- 2.3 Technical Background . . . 16
  - 2.3.1 Computation Offloading . . . 16
  - 2.3.2 Peer-to-Peer Network and WiFi Direct . . . 16
- 2.4 Our Approach . . . 17
  - 2.4.1 Approach Overview . . . 17
  - 2.4.2 Middleware . . . 18
- 2.5 Evaluation . . . 22
  - 2.5.1 Micro-benchmark . . . 22
  - 2.5.2 Case Studies . . . 23
  - 2.5.3 Discussion . . . 27
- 2.6 Related Work . . . 28
- 2.7 Conclusion . . . 29

3 RELIABLE AND EFFICIENT MOBILE EDGE COMPUTING IN HIGHLY DYNAMIC AND VOLATILE ENVIRONMENTS 30
3.1 Abstract 30
3.2 Introduction 31
3.2.1 Motivating Example 31
3.2.2 Technical Challenges 32
3.3 Technical Approach 34
3.3.1 System Architecture 34
3.3.2 Mobile Edge Services 35
3.3.3 Adaptive Edge Service for Reliability and Efficiency 35
3.4 Evaluation 40
3.4.1 System overhead 41
3.4.2 Case Study 41
3.4.3 Discussion 48
3.5 Related Work 49
3.6 Future Work and Conclusion 51
3.7 Acknowledgment 51

4 ENABLING FLEXIBLE AND EFFICIENT REMOTE EXECUTION IN OPPORTUNISTIC NETWORKS THROUGH MESSAGE-ORIENTED MIDDLEWARE 52
4.1 Abstract 52
4.2 Introduction 53
4.3 Background 55
4.4 Approach 56
4.4.1 Architecture Overview 56
4.4.2 Flexible and Efficient Group Organization 57
4.4.3 Remote Job Execution 59
4.4.4 Estimating Execution Time in Multiple Requests 61
4.4.5 API Usage Scenario 63
4.4.6 Failure Handling 64
4.5 Evaluation 65
4.5.1 Micro Benchmarks 65
4.5.2 Case Studies 68
4.6 Related Work 72
4.7 Conclusion 74

5 JSREX: AN EFFICIENT JAVASCRIPT-BASED MIDDLEWARE FOR MULTI-PLATFORM MOBILE PEER-TO-PEER NETWORKS 75
5.1 Abstract 75
5.2 Introduction 75
5.3 Background 77
5.3.1 WiFi-Direct 77
5.3.2 Code Offloading 78
5.3.3 Local Web Server 79
5.4 Approach 79
5.4.1 Architectural Overview 80
5.4.2 Java to JavaScript Conversion 81

5.4.3 Workflows 81
5.4.4 Run-time Considerations 85
5.5 Evaluation 89
5.5.1 Micro-benchmark 89
5.5.2 Case Studies 90
5.6 Related Work 97
5.7 Conclusion 98

6 INGRIM: A MIDDLEWARE TO ENABLE REMOTE METHOD INVOCATION ROUTING IN MULTIPLE GROUP DEVICE-TO-DEVICE NETWORKS 100
6.1 Abstract 100
6.2 Introduction 100
6.3 Background 103
6.3.1 Representative Technologies 104
6.3.2 WiFi Direct 104
6.3.3 ZeroMQ 106
6.4 INGRIM: Inter-group Remote Invocation Middleware 107
6.4.1 Service Definition 107
6.4.2 Proxy and Skeleton In Use 108
6.4.3 INGRIM Components 110
6.4.4 Function Call To Message Conversion 114
6.4.5 Message Flows 116
6.4.6 Group-to-Group Communications 119
6.4.7 Failure Handling 121
6.5 Evaluation 122
6.5.1 Micro Benchmark 122
6.5.2 Group-to-Group Communications 124
6.5.3 Devices To PC 125
6.5.4 INGRIM vs. RMI 126
6.5.5 Remote Motion Detection 128
6.6 Related Work 128
6.7 Conclusion 130

7 FUTURE WORK 131
7.1 Asynchronous WiFi-Direct Simulation 131
7.2 Compensation Model 132
7.3 Autonomous Reformation 132

8 CONCLUSION 134

REFERENCES 136

APPENDIX 147
A GRADUATE RESEARCH AND CREATIVE OPPORTUNITIES 148

CURRICULUM VITAE 151

LIST OF TABLES

Table Page

2.1 List of devices used in the experiments in chapter 2 22

3.1 List of devices used in the experiments in chapter 3. 41

4.1 List of devices used in the experiments in chapter 4. 65

5.1 List of devices used in the experiments in chapter 5. 89

6.1 List of devices used in the experiments in chapter 6. 122

LIST OF FIGURES

Figure Page

1.1 Image stitching service 5

1.2 Solution for dynamic and volatile edge computing. 6

1.3 The combination of edge computing and mobile networks. 7

1.4 Offload code in multiple mobile platforms 8

2.1 The overall system design. 18

2.2 Total energy consumption on a caller side. 23

2.3 Performance and energy consumption comparison of the image processing case study 25

2.4 Performance and energy consumption comparison of the Internet sharing case study 26

2.5 Experimental results comparison between remote and local GPS access. . . 26

3.1 Motivating example with image stitching service 32

3.2 Edge cloud architecture. 36

3.3 Aggregate overheads in two different networks. 41

3.4 The performance comparison of the edge server- and P2P-based execution model in image processing test. 43

3.5 Comparison of the estimated versus actual execution times in image processing test. 43

3.6 Word-counting: execution time and energy consumption. 44

3.7 Comparison of the estimated versus actual execution times in word counter test. 45

3.8 The edge server versus P2P-based service execution in internet share test. . 46

3.9 Execution time of the word count service measured in 3 different cases. . . . 47

3.10 System performance when requesting multiple services. 48

4.1 System architecture. 57

4.2 System detailed workflow 58

4.3 Brokerless and Broker-enabled models. 58

4.4 Average discovery and connection establishment time. 66

4.5 Time when sending various jobs to multiple devices. 68

4.6 Performance comparison of ZeroMQ and our middleware. 69

4.7 Sharing video across mobile devices 70

4.8 Image processing results initiating from the different devices. 70

4.9 Compare the cumulative execution time of the system between synchronous and asynchronous modes. 72

4.10 Compare the normal process when all components work smoothly and when failure happens in image processing test. 73

5.1 An overview of our middleware 77

5.2 The major components of our middleware. 80

5.3 The workflow that sends a request to the destination device. 84

5.4 The workflow that executes the remote task and return a result. 84

5.5 The workflow that returns a result to the requesting device. 85

5.6 The workflow that executes the remote task and return a result. 86

5.7 Performance comparison of multiple JavaScript engines on Android and Windows Phone platforms. 87

5.8 How JavaScript code is loaded by the default web browser and a local web server. 88

5.9 System overheads on two mobile devices when sending packages of varied sizes. 90

5.10 Discovery time and connection time on different devices 91

5.11 Image processing overview. 92

5.12 Compare the performance of the JavaScript and Native versions in image processing test. 93

5.13 Compare the performance of the JavaScript and Native versions with small data set. 93

5.14 Compare the performance of the JavaScript and Native versions with large data set. 94

5.15 Performance of PDF-2-Image converting service with 1 to 5 devices. 96

5.16 Compare performance of server offloading and P2P networks with 4 devices. 96

6.1 Performance comparison of WiFi and WiFi Direct. 105

6.2 Use of original WiFi interface to create group-to-group communications. . . 106

6.3 Simple use of the Proxy and Skeleton objects. 110

6.4 INGRIM components – Broker, FrontEnd, BackEnd, Sender and Receiver. . 111

6.5 INGRIM components – Bridge and BridgeX. 112

6.6 Sequences of the initialization process and message requests. 114

6.7 Message flow from a Service Proxy to the remote BackEnd. 118

6.8 Message flow from a BackEnd back to the FrontEnd. 119

6.9 Architecture of multi-group device-to-device communication with 4 devices. 120

6.10 Architecture of multi-group device-to-device communication with 4 devices. 120

6.11 A simple way to bridge device to PC. 121

6.12 Benchmark tests by running all components on one device. 123

6.13 Architecture of multi-group device-to-device communication with 4 devices. 124

6.14 An example of multiple groups. 124

6.15 A simple way to bridge device to PC. 125

6.16 Performance of device-to-device versus device-to-PC. 126

6.17 Comparing INGRIM and RMI in 4 test cases. 127

6.18 Performance of device-to-device versus device-to-PC. 129

LIST OF CODE SNIPPETS

Code Snippet Page
4.1 Overriding and initiating Requester 63
5.1 A sample of JavaScript template. 82
5.2 Functions of JavaScript Interface. 83
5.3 Loading an external JavaScript file using JQuery. 95
6.1 Service definition example. 108
6.2 Generated Proxy class. 109
6.3 Generated Skeleton class. 109
6.4 Example code to run Broker and Skeleton on the remote device. 110
6.5 Example code to run Broker, Bridge and a Proxy on the local device. . . . 111
6.6 BackEnd's service definition in JSON format. 113
6.7 Generated code in Service Proxy class. 115
6.8 Generated code in Service Skeleton class. 117

ACRONYMS

CORBA	Common Object Request Broker Architecture
DEX	Dalvik Executable Format
FCFS	First Come First Serve
FMMR	FFmpeg Media Metadata Retriever
G2G	Group-to-Group
GSN	Global Sensor Network
HTML	Hypertext Markup Language
INGRIM	Inter-Group Remote Invocation Middleware
JMS	Java Message Service
JSREX	JavaScript Remote Execution
LTE	Long Term Evolution
MANET	Mobile Ad-hoc Network
MNE	Microsoft Network Emulator
MOM	Message-Oriented Middleware
NIS	Network Information Service
P2P	Peer-to-Peer
PDF	Portable Document Format
PNG	Portable Network Graphics
RL	Responsiveness Level
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SM	Service Market
SMM	Service Market Middleware
TCP/IP	Transmission Control Protocol/Internet Protocol
WFD	WiFi Direct
ZMQ	ZeroMQ

CHAPTER 1
INTRODUCTION

Mobile-device users are continuously increasing their expectations relative to the functionality and quality of mobile applications. Meeting these expectations requires making use of sensory data, multimedia, and artificial-intelligence algorithms. Unfortunately, applications that incorporate these features tend to require inordinate amounts of computational power, battery energy, and network traffic, as well as extensive utilization of sensory resources. As a result, demands for new and more sophisticated functionality and higher quality is exceeding what is currently feasible on individual mobile devices.

Mobile applications often use cloud-based services as a means of enhancing the capacities of the mobile devices on which they run, both to improve the quality of service and to extend their functionality. However, accessing cloud-based services is not always feasible, beneficial, or safe [9, 10]. First, in many circumstances cloud-based services may not be reachable via high-quality network connections, thus negating their potential benefits [11]. Second, network communications are energy intensive, so relying too heavily on cloud-based services could unnecessarily drain battery power [12, 13]. Third, remote network connections can become overwhelmed in the presence of multiple concurrent users, as is often the case in crowded public places like airports and large events. Finally, in some environments, a user may not feel safe accessing any remote services by means of unsecured or unknown access points [14–16].

With the rapid growth in number of mobile devices, their collective computational power could provide an alternate solution for distributing heavy computational workloads. For example, consider a mobile application that needs to run image processing on a series of large-scaled images, each with a size of 4000×5000 pixels. A device would have to allocate approximately 60MB for one image and approximately twice that for temporary buffers and the results. Currently, the Android system does not allow an application to allocate that amount of memory and will terminate any application that tries. A solution to this design problem could be a middleware system that allows the mobile device to offload computation to peers. However, offloading creates a new problem, namely how to select the most suitable devices for offloading such that the system as a whole ensures high performance and low energy consumption in opportunistic networks comprised of arbitrary mobile devices. The challenge of selecting appropriate peers problem has been a fundamental research problem for the last decade [17].

Large-scale or massive face recognition is another example. Detecting multiple faces from a large population would require an application to load the subject image into memory, identify each face, compute its characteristics, and then try to match those characteristics with potentially hundreds of thousands of known face profiles. Furthermore, if the full database of face profiles is too large to reside on the mobile device, the application would have to download some or all of the profiles from a remote server on the fly. Even if the subject image was small enough to fit load into memory, it would still take a mobile device an unreasonable amount of time to complete the necessary computations and comparisons. This design problem could be solved by employing an edge computing architecture where edge servers are scattered and cache the profile database for handling matching computations of the nearest devices. Although this approach could significantly improve the matching speed compare with the tradition clouds by caching profiles, it is still exposed to high volatility, especially when the devices are moving out of contact range with the edge servers.

A third example of when mobile applications could benefit from offloading computations to peer devices occurs during disasters or disaster relief efforts. In a disaster area, the network infrastructure can be severely damaged or completely destroyed, so devices cannot access cloud-based services directly. This problem could be solved if mobile devices could run a mobile application that creates an inter-network across ad-hoc peer networks and that could span the disaster area until some devices on the edge establish Internet connections [16]. Like the first example, this solution creates a new problem of selecting the most effective devices to act as routers between groups of peers.

This dissertation presents, through a series of scholarly articles, a middleware system called, UMSEF – an Universal Mobile Service Execution Framework, that enables mobile applications to establish opportunistic inter-networks and then transparently share computational tasks and network resources over those networks. In particular, in creating UMSEF – an Universal Mobile Service Execution Framework, the research addresses the following series of technical problems: (1) the effective partitioning and distribution of tasks to nearby peers based on their current resource availability, (2) designing a software architecture for edge platforms that is capable of detecting system malfunctions and network failures and support a degree of self mending, (3) designing portable framework that enables the offloading of general-format packages, (4) facilitating the use Remote Method Invocation in mobile applications through code generator to reduce development time and improve distribution over multi-group device-to-device networks, and (5) extending the limited range of traditional WiFi-Direct-based communication through inter-group connectivity, while optimizing the speed of dispatching requests for real-time applications.

1.1 Task Partitioning And Distribution

For the first problem, we developed a middleware framework, based on WiFi-Direct, that allows a mobile application to distribute jobs to nearby devices based on their current resource capacity and availability. A mobile application can use this framework to effectively partition large tasks into jobs and offload them to peers, without having to directly deal with the peer selection and communication issues. Specifically, before dispatching job requests that comprise a large task, a framework sends *resource request* messages to the other devices in the group, then collects *resource feedback messages* from them. Each resource feedback message contains percentages that represent the availability of peer device's CPU, RAM, battery and sensors, as well as a *responsiveness level* that represents its overall responsiveness via the connecting network. Then, the framework compares those values with the same kinds of value for host device and filters out those peer devices that do not currently have sufficient availability or responsiveness to participate in the task. The framework then distributed the jobs that comprise the task to the remaining peers. According to our experiments using a testbed with real devices and a variety of mobile applications, when the framework was used to distribute the workload across multiple devices, most of applications exhibited better performance than when they were run on a single device. Depending on the application, i.e., the type of task being distributed, and number of peer devices, the speedup was as high as 70% and reduction in energy consumption as high as 50%. (See Chapters 2 and 4).

1.2 Highly Dynamic And Volatile Edge Systems

The second problem deals with edge computing, which tries to exploit data locality by processing massive amounts of sensory data collected by IoT and mobile devices ”at the edge of the network.” Edge servers process data collected from nearby devices and transmit only results to the cloud [18, 19]. They can also cache data from the cloud for use by nearby devices and can coordinate at-the-edge computation by assigning tasks to the available connected devices [20]. To date, most systems that utilize edge computing operate under the assumption the network connections are stable, both between the edge servers and the cloud and between local devices and the edge servers. In our research, we only consider edge-computing environments that are *highly dynamic* and *volatile* [21]. These environments are characterized by intermittent network connectivity, device mobility, and the presence of partial failure. In other words, the network connections both within the edge cloud and to the Internet are unstable or even non-existent. Users carrying the mobile devices move at will, thus potentially affecting their devices' reachability to the edge cloud. Also, any other devices involved, including the edge servers, can crash or become unreachable at any time.

For example, consider a large-scale agriculture system that uses autonomous tractors to gather multi-spectral imagery on multiple fields and then processes those images to detect areas damaged by destructive insects, insufficient watering, toxins, or poor soil composition. Assume that each field has an edge server and that tractors have intermittent WiFi connections to these edge servers. Also, assume that the edge servers periodically have cellular data connections to the Internet. When a tractor has a connection to an edge server, it can dump its imagery data to that server, which can then process that data and send the results to a central server when it has an Internet connection. In this system, there is a high probably that the tractor may enter area without WiFi coverage for any extended period of time and could, therefore, fill up its local memory and start to lose data. Similarly, it is possible that an edge server is unable to establish an Internet connection for an extended period of time, so the central server may have gaps its data. Both of these problems could be addressed by middleware that enables inter-networking between tractors and edge server, or between edge servers, over opportunistic ad hoc networks comprised of tractors.

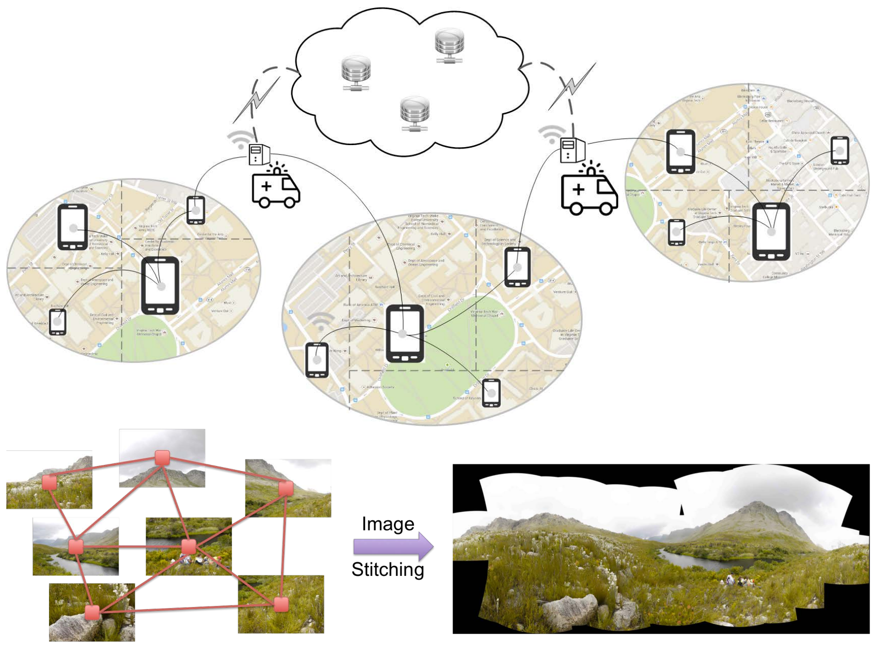


Fig. 1.1: Image stitching service

Another example is communications among first responders in disaster area, consider Figure 1.1. Each responder is supported by a suite of personal devices, both mobile and wearable. These devices are heterogeneous, in the sense that they differ in terms of their hardware resources, operating systems, and platform versions. A recovery vehicle hosts an edge server that also provides a WiFi access point (AP). Assume that the edge server's processing power is vastly superior to those of the personal devices and the device's cellular signal is intermittent or non-existent. Enabling the mobile devices and the edge server to cooperate as an edge cloud can greatly assist the responders in their mission, which is to assess the situation and come up with a recovery strategy (Figure 1.2).

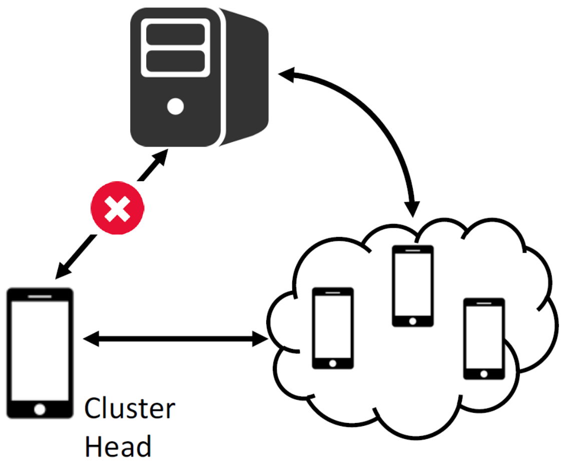


Fig. 1.2: Solution for dynamic and volatile edge computing.

For these and other systems, to realize the vision of enabling the heterogeneous mobile devices and the edge server to collaborate as a coordinated edge cloud, it is necessary to address two key technical challenges: device mobility and partial failure. Mobility is an inherent requirement for many applications, like the two mentioned above. Even when devices remain within signal range, communications are likely to exhibit a high degree of volatility, latency, and packet loss rates. The more serious problem is that they can move out of signal range, thus network partitioning or partial failure – which is the second key technical challenge. With an appropriate middleware solution, partial failures can be addressed by automatically re-routing communications over over peer-to-peer connections and reassigning the execution of certain jobs to reachable devices.

To support highly dynamic and volatile edge-computing environments, we provide a *service infrastructure for reliable and efficient mobile edge computing* that includes adaptive facilities to dynamically restructure a distribution pattern in response to partial failure.

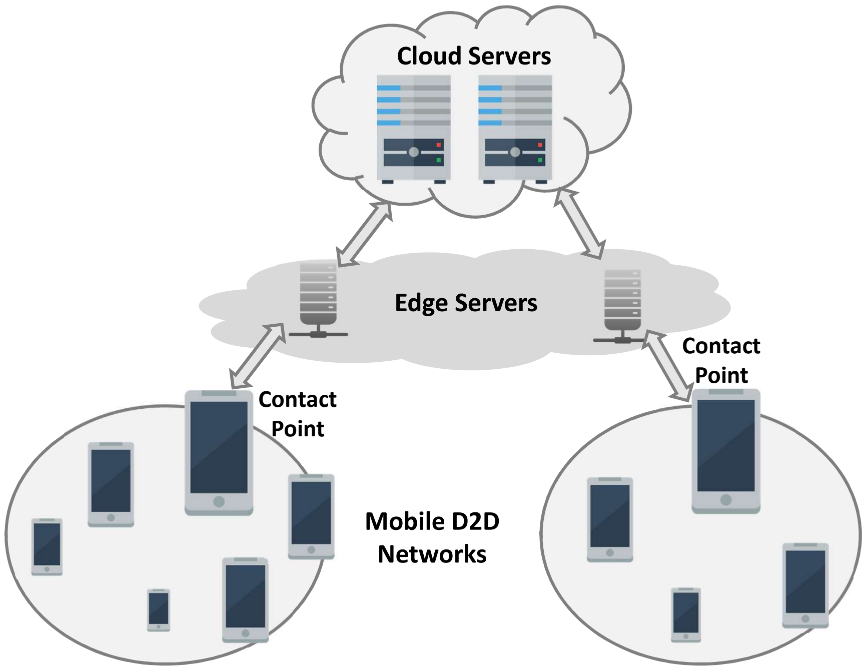


Fig. 1.3: The combination of edge computing and mobile networks.

The primary service-execution model is a client-server model with micro-services at the edge server, as long as it is reachable (Figure 1.3). Clients maintain two communication channels with an edge server and a *Cluster Head*. By exchanging a heartbeat message between an edge server and clients, each client can check the availability of the edge server and network status to both the edge server and nearby devices. If a client does not receive an acknowledgement from the edge server due to the network disconnection, it informs nearby clients of the network failure through a cluster head. Then, clients immediately switch their service execution model to the P2P mode and continue the failed service execution through a peer-to-peer network. Then, as soon as the network connection between the edge server and mobile devices is restored, the service execution model is switched back to the client-server model. (Chapter 3 and Chapter 5).

1.3 Offloading Of General-Format Packages

The third problem addresses the ease with which mobile application developers can create distributed systems and the systems' portability across common mobile platforms, e.g., Android, iOS, and Windows. Ideally, developers should not have to deal with any details related to forming opportunistic networks, identifying and selecting peers, managing the distribution of jobs and collecting responses. In fact, the use of remote computational resources should be entirely transparent to application developer.

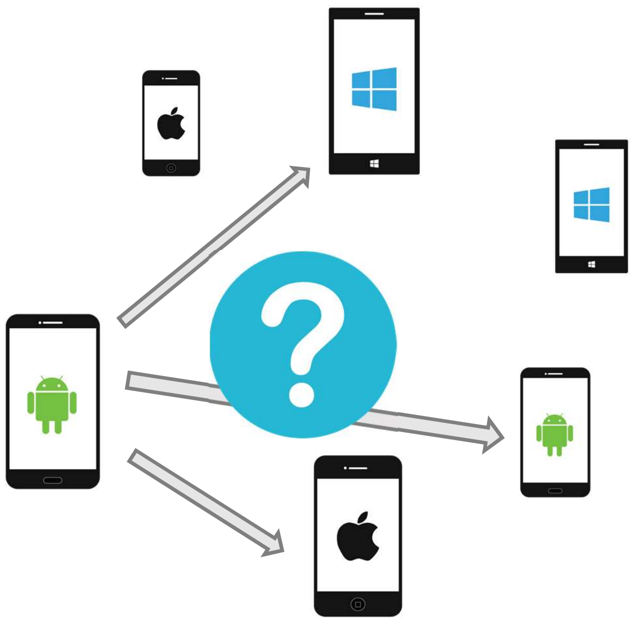


Fig. 1.4: Offload code in multiple mobile platforms

Although, it is valid to assume that peer devices may have some middleware installed and running, it is critically important that they are not required to download and install specific applications in advance of their use in offloading. This means that job distribution needs to include both the code to be executed and the data that the job requires. Furthermore, these jobs need to be portable across platforms so any mobile device may participate in the system.

Although code offloading in device-to-device networks has been researched for years [22, 23], its use in the industry is still modest. One reason for its slow adoption is difficulty of cross-platform support. Consider a camera-sharing application that collects images from nearby devices in real time, then stitches those images together to create a 3D image. The application may not know the type of devices (and their camera characteristics) ahead of time, nor will it be able to detect them due to privacy issues. So, the application must be able to work with generic and portable camera or image interfaces. To date, many code offloading architectures have been proposed but they only targeted a single specific platform [3, 4].

A straightforward solution for the cross-platform problem is using JavaScript to write code and distribute to the peers. For example, Migratom.js [24] is a code migration system that uses a flow-based paradigm to accelerate mobile web applications by offloading compute-intensive tasks to the superpower servers. JSCloud [9] invokes the code analysis and instrumentation phases to decide whether to start offloading and which code partitions to offload to the cloud. JSCloud supports a wide range of devices and computers, but the estimation relies on interpolation which incurs more computation cost. Another approach [25] analyzes offloadable code, written in JavaScript, to enhance performance of web applications. However, improvements in speed with this approach are limited to JavaScript parts of web applications and the offloaded code must run in a cloud environment [25, 26]. In opportunistic networks where cloud servers are not available, the native app must equip itself with an appropriate JavaScript engine to receive and execute the JavaScript code, be able to inject data into JavaScript environment, and bubble up responses from there to the native layer.

The paper presented in Chapter 5 addresses these problems and provides a workable solution by introducing a technique for automatically converting application code (in this case Java code) to JavaScript and then encapsulating it with data and meta data to form a portable job request message.

1.4 Flexible N-N Model For Device-To-Device Networks

In the mobile-computing domain, the WiFi-Direct technology has received much attention as a means for creating opportunistic networks [27]. This technology allows a mobile device to discover and establish connections with other devices within WiFi signal range without a standard *access point*. It establishes an isolated network group by electing one device as a *Group Owner* (GO) and to act like a virtual access point with a DHCP service. It assigns IP addresses (range of 192.168.49.x) to itself and other devices in the group. Because the groups are isolated, WiFi-Direct only provides 1-to-1 or 1-to-N communications, which by itself, minimizes its application for offloading in dynamic edge environments where more complex and flexible network communications are needed.

Some other research has addressed this problem by providing workaround solutions. For example, one approach, called IP Subnet Negotiation Protocol (ISNP) for Seamless Multi-Group Communications [28], runs before the establishment of groups to allow any subsequent multi-grouping protocol to succeed in creating bidirectional links between the groups [1]. Another approach, called Efficient Multi-group formation and Communication (EMC) [2], exploits the battery specifications of the devices to qualify potential group owners and enable dynamic formation of more efficient groups. It utilizes the service discovery feature of WiFi-Direct to allow devices to share their battery information. A device with a richer energy reserve than those in its range opts for creating a group. Once a group has been formed, the owner designates from among its members what is referred to as proxy members (PMs) that link the group to other groups. Similarly, Casetti et al. used the *Legacy Client*[2] to bridge from the member of a group to GO of another group, for data-centric networks [29]. However, these solutions are limited because they either require modification to the operating system (e.g. device rooting), lacked open APIs for developer to integrate in their distributed systems, or are overly complicated to integrate into mobile applications and deploy on heterogeneous platforms.

1.5 Enabling RMI On Mobile Platforms

Code offloading is known to have slow responsiveness due to high overheads and permission checks [30]. The architectures designed for code offloading or code migration are inappropriate to apply for real-time applications. For the systems with high demand of

[1]ISNP takes advantage of the service discovery mechanism in WiFi-Direct to allow a participating device to negotiate distinct IP subnets with other devices in a way that does not require them to be connected by any other means. Once multi-groups have been created by subsequent protocols, each GO uses its proposed IP subnet [28].

[2]The term Legagy Client refers to the original WiFi adapter or network interface

rapid distribution, Remote Method Invocation (RMI) framework or CORBA model seems to be a better fit as they allow invoking remote method calls on local object on top of transparent network layer. However, these two technologies are out-of-date for the time being, their unwieldy structures require a stable network infrastructure which does not suit mobile networks characterized by intermittent connections, their APIs are deprecated on lightweight mobile platforms such as Android as well.

Some attempts to address this problem by providing different solutions for Android RMI [31, 32], which are lightweight RMI packages dedicated for Android platform, these approaches leverage the *original Binder* to allow users to invoke system services as well as application services between devices using *remote parcel* format. Using external intents as messages, they abolished the cumbersomeness of the traditional RMI and made it adaptable with unstable networks established among the devices. Nevertheless, these approaches expose two major problems: (1) *platform dependence*: due to the reliance on *Binder* which is only available on Android OS. To adapt with the other platforms, an adaptive layer must be built in between to serialize Binder objects to binary array for network transmission and deserialize back on another platform. (2) *Tightened with 1-1 (or 1-N) communication model*: using Binder limits the choice of network topologies and raises the complexity for extension to larger scale network models such as N-N or multiple groups.

Inter-group network can significantly extend the distance limit of WiFi-Direct over 200 meters. To support inter-group, we utilized the idea of *Legacy Client* [29] which is a *bridge* to open connection between one device from one group with the owner of another group on *original WiFi interface*. We built our middleware following component-based which include 4 out-of-the box components: *FrontEnd* for dispatching requests, *BackEnd* for handling processes, *Broker* for navigating requests to destinations and *Bridge* for inter-group, each combination of these building blocks can construct different network topology. To simplify the software integration, we provide a function-scope declaration mechanism for developer to define service by annotating on function prototypes, then these functions will be automatically converted to service during the code compilation. The converted services handle the remote calls by serializing and wrapping them into binary packages then forwarding them to the peers (Chapter 6).

1.6 Offloading In Multi-Group In Mobile Applications

Finally, we perceived the possibility of combining remote method invocation and code offloading. As mobile device is characterized as a personal device, it faces everyday threats such as personal information being collected, device resources (e.g. camera, CPU or RAM) being taken advantage of, or credit card information being stolen. Therefore, running an external process on a mobile device which has not passed the background permission checks is always considered as high risk, making code offloading technology arduous to be incubated in the industry. According to this problem, one solution for the mobile devices that are willing to adopt external programs is: before the process execution, they must all preliminarily agree on a function list and accompanied permissions. To do so, the device must register to receive a *function prototype* [3] reflecting the actual service so that it can safely offload through the prototype in the same way RMI model does. Moreover, due to the wide-range distribution and mobility of mobile devices, the system should address three important requirements: intermittent communications, inter-group networks, device filter and selection.

Each *BackEnd* registers a set of service functions to the nearest *Broker* at the system initialization, this registry will be routed by the *Bridges* to the other *Brokers* for synchronization in inter-group network. To ensure the requesting device will receive either the final result for a request or an accurate error message from the system if any problem occurs on message route, we provided a fault handling mechanism which considers message *time-out* value to evaluate the success of every transmission step on each component. If error happens, an error message will be routed back to the requesting source.

1.7 Summary

The five problems mentioned above are central to creating portable middleware and

[3] In RMI the function prototype is called *Stub* class

programming frameworks, namely UMSEF, which aims to support seamless code offloading in dynamic edge environments. The papers found in the next five chapters address one or more of these problems and collectively establish a foundation for UMSEF – an Universal Mobile Service Execution Framework. Because these papers were written over a period of two years, some of the names and terminology have changed, but their contributions represent incremental steps towards the overarching offloading problem. Also, even though paper contributions are significant, they do not represent a final solution and there is still work to be done. Chapter 7 discusses future research directions and ideas for unifying the middleware and framework components discussed in the individual papers into a cohesive, open-source package. Finally, Chapter 8 provides some general conclusions about the work presented here and its value to the mobile community.

CHAPTER 2
UTILIZING NEARBY COMPUTING RESOURCES FOR RESOURCE-LIMITED MOBILE DEVICES

[5] [1]

2.1 Abstract

For the last decade, mobile devices have been significantly developed with powerful hardware facilities such as multicore CPUs, large and fast memory, fast network and high-resolution displays. Moreover, mobile applications deliver increasingly complex functionality thereby still requiring ever greater hardware capability. As a result, overcoming resource limitations in mobile software development has become a major research challenge. To that end, in this chapter, we present a distributed execution infrastructure that enables mobile applications to access remote resources (e.g., CPU, memory, network, sensors, etc.) based on two distribution models: client/server and peer-to-peer. We break down a remote execution into small execution units (e.g., code and data) based on the availability and resource capacities of nearby devices. The benchmarks and case studies demonstrate that the existing mobile devices can extend their resource capabilities through our distributed execution infrastructure, so that it makes possible to introduce new hardware capacity (e.g., sensors) to existing mobile devices as well as increasing both performance and energy efficiency of mobile applications.

2.2 Introduction

Mobile devices have been evolving at a lightning pace with powerful hardware facilities such as multicore CPUs, large and fast memory, fast network and high-resolution displays [33]. Due to the significant development of mobile hardware, today's mobile applications are

[1]Minh Le, Young-Woo Kwon @ SAC 2017

becoming more complex with an increasingly feature-rich nature. As a result, mobile devices often overtake the personal computer as a primary means of accessing computing resources. However, the resource demands of mobile applications often outstrip the hardware capacities of mobile devices. A particularly popular technique to extend limited mobile hardware is *computational offloading* executing CPU-intensive functionality at a powerful cloud-based server, thereby improving both performance and energy efficiency.

Although computational offloading has been widely enjoyed in the research literature [34, 35] as an optimization mechanism that can utilize remote CPU resources, a majority of feature-rich mobile applications still suffer from resource limitations to provide quality user experiences. In addition, because computational offloading mechanisms have been developed by leveraging cloud computing technologies, despite their significant advantages, the high operational cost of cloud infrastructures and the implementation difficulties of computational offloading have deterred programmers from actively applying computational offloading in their mobile applications [36]. Furthermore, performance or energy benefits gained through computational offloading would be considerably low when comparing to the operational costs of the cloud-based offloading server. Finally, implementing effective computational offloading optimization often requires highly experienced programming skills and efforts.

Nevertheless, offloading-based mechanisms are still considered an important optimization technique for mobile applications [37, 38]. Thus, in this chapter, we present a novel distributed execution model that provides two offloading models: client/server and peer-to-peer model to optimize mobile applications executions in terms of performance and energy efficiency but also extends mobile devices hardware capability. Mobile devices will be able to add virtually more hardware resources including computation, networking, memory, and sensors to the existing hardware setups. Specifically, our middleware system determines an offloading strategy between client/server and peer-to-peer model and then distributes the requested execution over the nearby or remote devices [39]. The middleware system employs a dynamic, adaptive mechanism to determine the best distribution and execution strategy under different execution environments including diverse network conditions and mobile devices.

The experiments in three case studies have demonstrated the effectiveness of our approach, to extend limited mobile hardware resources, thereby improving performance and energy efficiency as well as bringing new hardware capabilities. The rest of this chapter is structured as follows. Section 2 introduces a technological background for the main technologies used in this work. Section 3 details our technical approach and Section 4 discuss how we evaluated our approach. Section 5 compares our approach to the related state of the art. Section 6 concludes this chapter.

2.3 Technical Background

In this section, we present major technologies including cloud offloading, peer-to-peer and WiFi Direct technology.

2.3.1 Computation Offloading

Computation Offloading-executing computation intensive functionality and receiving a result only-has become a popular optimization technique for mobile applications [4], [40]. It leverages the resources of cloud-based remote servers to execute portions of a mobile applications functionality. By executing some of the applications functionality in the cloud, offloading reduces the amount of energy consumed by the mobile device, thus saving its battery power. An additional benefit of computation offloading is improved performance efficiency, as cloud servers have hardware resource more powerful that those available on mobile devices. This technique is used as one of the distributed execution models in this chapter.

2.3.2 Peer-to-Peer Network and WiFi Direct

WiFi Direct [2] is a new peer-to-peer standard built on top of the IEEE 802.11 to provide direct connections between WiFi devices without an Internet connection. Over a WiFi

[2]Wi-Fi Direct: http://developer.android.com/guide/topics/connectivity/wifip2p.html.

network, a device can discover and connect to any devices without special configurations or setups. Once a connection is established, the devices can communicate with each other as a client or a group owner. WiFi Direct has been widely used to share media contents between mobile devices. Our approach uses Android WiFi Direct to use near computing resources without an Internet connection or a wireless access point (AP).

2.4 Our Approach

Next, we present our approach, a distributed execution model that can extend the resource capacities of a mobile device. We start by giving an overview of the approach and then describe its major parts.

2.4.1 Approach Overview

Our approach provides a communication and collaboration infrastructure between the nearby devices. Figure 2.1 shows the overall system design. Specifically, `JobExecutionManager` determines appropriate distribution units and targets and then steers job executions through `DataParser` that splits a job into small data chunks. Prior to dispatching a job to peer devices, `JobDispatcher` first discovers available mobile devices by sending a broadcast message to nearby devices and then calculates the availability of peer devices. For the optimal execution result, `JobExecutionManager` keeps track of current execution environments such as network conditions (e.g., delay, bandwidth) and system status (e.g., resource usage including CPU, memory).

On a peer side, `JobClassLoader` instantiates the job class and executes an entry function using the attached data chunk. Then, the caller waits for the execution result from each peer until receiving all the results for a certain time-out period. Otherwise, they are considered missing results and re-executed through fault-handling mechanisms. Moreover, `JobExecutionManager` keeps monitoring the execution status of peers, so that if there is any fault during the execution, `JobExecutionManager` immediately restarts the failed job locally. In addition to the exception handling mechanism, we use the checksum to verify data integrity.

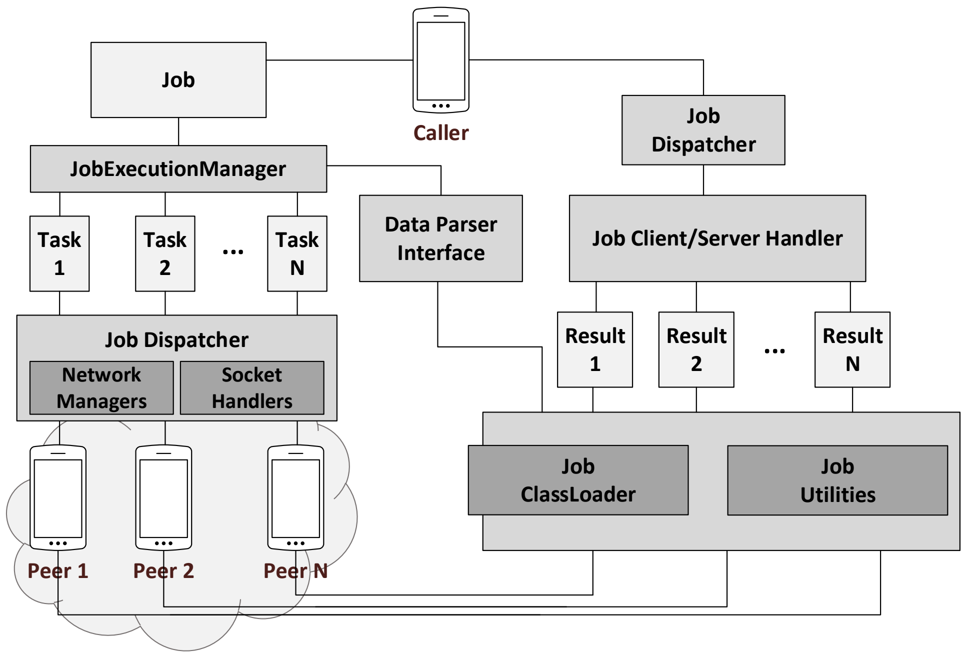


Fig. 2.1: The overall system design.

2.4.2 Middleware

In this section, we discuss (1) how to discover available peers and exchange their information, (2) how to select appropriate peers, and (3) how to partition data to be sent to the selected peers.

Discovering Available Peers

During the initialization phase, our middleware identifies available peers by broadcasting a status inquiry message to the nearby peers. Upon the receipt of the message, the middleware immediately sends back a response message, which contains the following information: *Responsiveness Level* based on the current resource usages, the current battery level, GPS and network availability, etc. In particular, the *Responsiveness Level* is a parameter that represents how favorable the selected peer is to execute the requested job. In the following discussion, we show how to compute *Responsiveness Level*.

Estimating Resource Usages

Since *Responsiveness Level* (`RL`) represents devices resource availability, it is proportional to the resource capacities (e.g., the number of cores, CPU speed, total memory) and their usages. As a result, the more resources consumed, the more responding time the peer would take. To select most suitable peers, `RL` is calculated on each nearby device as follows:

$$RL = \frac{N_{Cores} \times CPU_{Speed}}{Usage_{CPU}} + \frac{Mem_{Spec}}{Usage_{Mem}} + \frac{Batt_{Spec}}{Usage_{Batt}} \tag{2.1}$$

where N_{Cores} is the number of CPU cores. CPU_{Speed} is the speed of a single core in GHz. Mem_{Spec} is the memory capacity in GB. $Batt_{Spec}$ is the battery capacity in Ah. $Usage_{Mem}$ and $Usage_{Batt}$ are the current memory and battery usages, respectively.

The higher value of `RL` means higher availability. Thus, if the selected peer is a powerful cloud-based server connected to a wall power outlet, $Batt_{Spec}$ is ∞ and $Usage_{Batt}$ becomes 0, resulting in ∞ for `RL`. In our design, the `RL` value is calculated on each peer in advance and then sent back to the caller before the job assignment, thereby reducing the runtime overhead on a caller side.

In the following discussion, we present how each resource usage information can be collected and quantified.

CPU Information While CPU frequency (CPU_{Speed}) can be easily obtained by reading the `cpuinfo_max_freq` system file, the ratio of CPU usage is calculated using the `/proc/stat` system file as follows:

$$Usage_{CPU} = \frac{(\sum T_{CPU2} - T_{Idle2}) - (\sum T_{CPU1} - T_{Idle1})}{(\sum T_{CPU2} - \sum T_{CPU1})}$$

Where $\sum T_{CPU_i}$ and T_{Idle_i} are the total active time and idle time in two consecutive inquiries.

Memory Information For the memory usage information, we also collect the total memory capacity and current usage information. In particular, the total memory capacity and usage information can be obtained using the `MemoryInfo` class In Android. `MemoryInfo` provides `availMem` property for actual available memory in MB (denoted as $availMem$), and `totalMem` for total accessible total memory in MB (denoted as Mem_{Total}). The ratio of available memory can be calculated as follows:

$$Usage_{Mem} = 1 - \frac{availMem}{Mem_{Total}}$$

Battery Information

Next, the battery information including the total capacity ($Batt_{Total}$) and the current usage ($Batt_{Used}$) can be collected from the `PowerProfile` class and `ACTION_BATTERY_CHANGED` service. Then, we calculate the ratio of available battery as follows:

$$Usage_{Batt} = \frac{Batt_{Level}}{Batt_{Total}}$$

Where $Batt_{Total}$ is the battery capacity in mAh and $Batt_{Level}$ is the current battery usage reported by an operating system.

Selecting Available Peers

Next, depending on the amount of data, appropriate peers are chosen for the best execution result. To that end, we take multiple parameters representing execution characteristics (e.g., CPU-intensive computation, sensor access), data size, performance/energy prediction, and peer status into consideration. In addition, computational offloading [41], [34] has received much attention as an energy/performance optimization mechanism. Thus, we will take advantage of a powerful cloud-based remote server as a peer. To select an appropriate number of peers, we use a heuristic way as follows:

- *If only one peer is requested* (e.g. request GPS or sensor data): Select one peer that has the largest RL value and meets other criteria.
- *If an offloading server is available* (e.g., CPU-intensive job): Select the offloading server and ignore nearby devices only the network condition is favorable.
- *Otherwise* Select nearby peers based on RL that have larger RL than the callers one.

Partitioning Data

After identifying available peers, the run-time system partitions data that will be sent along with a job. As a result, the runtime system can ensure that each peer receives an appropriate amount of job that is suitable to execute. Specifically, the amount of data sent to peers is calculated as follows:

$$M_i = M \frac{RL_i}{\sum_{j=\overline{1,n}} RL_j} \tag{2.2}$$

where M is the total size of data in bytes, n is the number of active peers, and j-peer has its *Responsibility Level*, RL_j, respectively. If an offloading server is available, we calculate the amount of data sent to the server $M_{offload}$:

$$M_{offload} = M \frac{RL_{offload}}{\sum_{j=\overline{1,n}} RL_j} = M \frac{1}{1 + \frac{\sum_{j \neq offload} RL_j}{RL_{offload}}}$$

From Section 2.4.2, we know that $RL_{offload}$ is ∞ for an offloading server. Since $\sum_{j \neq offload} RL_j$ is limited due to peers' resource constraints, $M_{offload}$ becomes M. As a result, we can flush the whole task to an offloading server to gain the best result. Based on these estimations, we assign appropriate amounts of data to each peer through the following steps:

- Calculate the total size of data (M) and the responsiveness levels ($\sum_{j=\overline{1,n}} RL_j$) of all peers.
- Estimate the data size to be sent to each peer (Mi) and then partition data into smaller parts.
- Produce n distribution units containing an extended Job class and data.
- Dispatch each distribution to each peer.

Handling Run-time Failures

Due to the nature of volatile mobile networks, partial failure may occur because each component of a distributed execution (e.g., peers, offloading server, or the network) can fail independently. Thus, to ensure all jobs are completely executed at peers and their results are returned to a caller, our run-time system listens to network-related updates (e.g., network join/leave events from the `BroadcastReceiver` class) and as well as catching exceptions thrown from the underlying system (e.g., TCP socket, WiFi Direct Controller, Android, etc.). Moreover, we use a simple checksum mechanism to check the integrity of an execution result. If any types of execution failures occur, the `Execution Manager` module immediately re-executes the failed job on a caller.

2.5 Evaluation

We evaluated the effectiveness of our approach in improving energy- and performance-efficiency as well as introducing a new hardware capability to resource-limited mobile devices. The experimental setup includes a test-bed with five Android phones and one cloud-based server (Table 2.1). Then, we have experimented with three emulated network conditions described in Section 2.5.2. To measure energy consumption, we used a Monsoon Power Monitor device [3]. For the offloading server, we set up an Android-based server using Android x86 [4]. We only measured energy consumption and time at a caller side.

Table 2.1: List of devices used in the experiments in chapter 2

Devices	**CPU**	**RAM**	**Battery**	**OS**
LG Opt. GK	1.7GHz	2GB	3100mAh	4.4.2
LG G Stylo	1.2GHz	1GB	3000mAh	6.0
LG Tribute	1.2GHz	1GB	2100mAh	4.4
LG G4	1.44GHz	3GB	3000mAh	5.1
Galaxy S3	1.4GHz	1GB	2100mAh	4.4
Asus Zenfone 2	1.7GHz	3GB	2400mAh	5.1
LG G Pad	1.7GHz	2GB	4600mAh	4.2.2
Lumia 550	1.1GHz	1GB	2100mAh	W. 10

2.5.1 Micro-benchmark

We first performed a micro-benchmark to evaluate the overhead introduced by our

[3] Moosoon Power Monitor: https://www.msoon.com
[4] Android x86: http://www.android-x86.org

middleware including constructing a peer-to-peer network, monitoring peers status, and steering job executions. Figure 2.2 shows the amount of energy consumed to maintain the peer-to-peer network. Surprisingly, the number of peers does not affect the total energy consumption of a caller. Because our approach transmits the same amount of data regularly over the WiFi network, from the statistic information we collected, the energy variability of one device within P2P network in idle condition can be simply represented by a linear function.

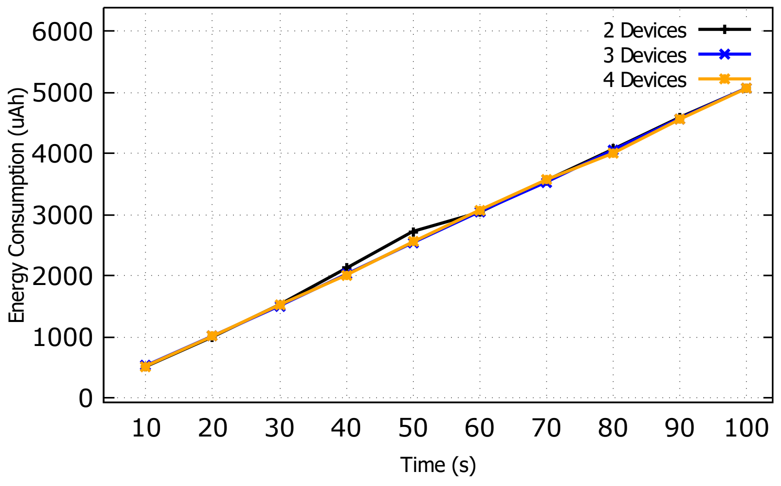


Fig. 2.2: Total energy consumption on a caller side.

2.5.2 Case Studies

Then, to evaluate the performance and energy efficiency of the our system, we experimented with three case studies:

- **Image Processing** An image is split into several parts, each of which is blurred through nearby devices.
- **Internet Sharing** A device without an Internet connection utilizes the dearby device network resources to download a web page.

- **GPS Sharing** A device requests location information to nearby devices that have locality functionality (e.g.,GPS, network provider, etc.).

Image Processing Application

We first applied our approach to an image blurring application requiring complex image processing. We experimented with the benchmark application with two images that have different sizes: the large image has 4326×2856 pixels and the smaller ones size is 2500×1405 pixels. To distribute the image to nearby devices, the caller splits an image into a number of vertical portions, the size of which is proportional to their `RL` value.

First, we experimented with a large size image, which needs about $50MB$ memory space to load and another $50MB$ for the result. A low-end mobile device may not load and process the image due to the resource limitations. Thus, when executing the application on multiple devices through our approach, we reduced the execution time by 2345% and increased the energy efficiency by 3545%, respectively. For the small size image, our approach could run 3552% faster and save 20-33% more energy than running on a single device. Figure 2.3 shows the experimental results.

Internet Sharing Application

Next, we implemented an Internet sharing application that can accelerate downloading Internet contents through nearby smartphones. A request consists of URL, n (i.e, number of peers) and index (i.e, device order). Then, each device (including the caller) downloads the HTML text contents from the URL, collects its resource URLs (images, videos, audio etc.), and downloads a batch of those URLs according to its index position of n devices.

As depicted in Figure 2.4, we could reduce the downloading time by 62%, as well as saving 63% of energy consumed as compared to downloading the same contents on a single smartphone.

GPS Sharing

Although acquiring a GPS location is high energy consumption process, our approach

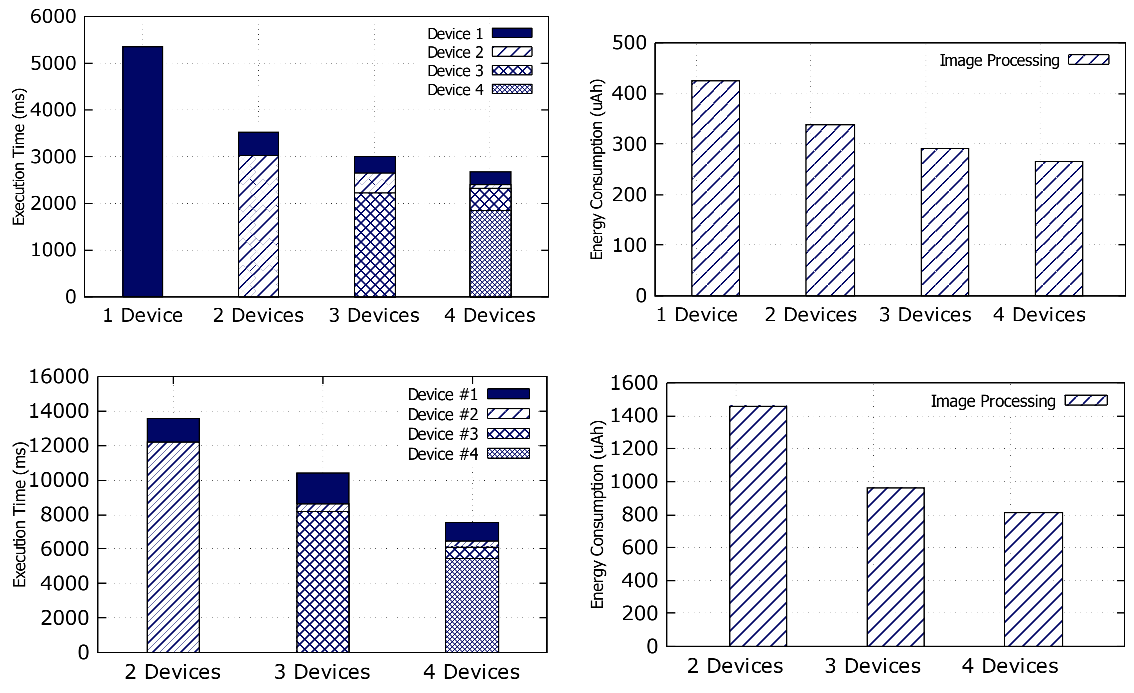


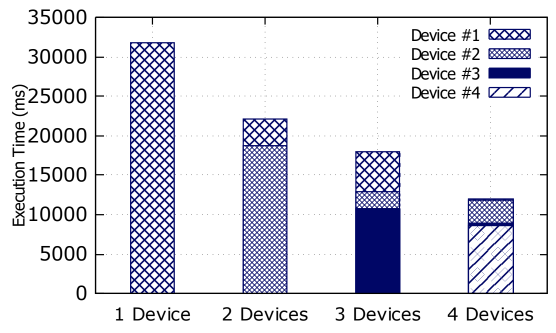
Fig. 2.3: Performance and energy consumption comparison of the image processing case study

allows a device to request the GPS location from a healthier device. To that end, we only select a device that has a GPS and the largest value of RL (i.e., the most suitable device).

We experimented with two test cases: remotely and locally obtaining GPS locations. As shown in Figure 2.5, the GPS sharing functionality has the similar execution results in terms of energy efficiency and performance. Thus, GPS needs to be accessed locally for energy efficiency and performance. However, it makes possible to provide location information to mobile devices that are not equipped with GPS or users who dont want to reveal their actual location for the privacy reason. As a result, our approach enables a mobile application to utilize virtual resources.

Comparison with Cloud Offloading

Finally, we compare peer-to-peer- and cloud offloading-based executions in terms of performance and energy efficiency to determine where and when to offload jobs.



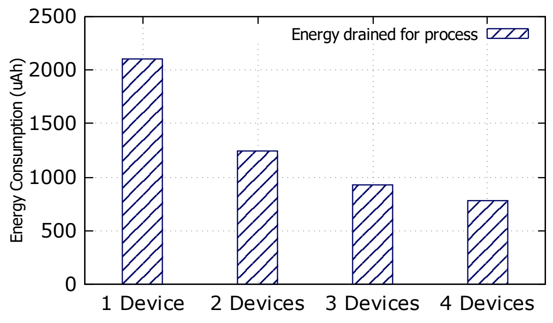
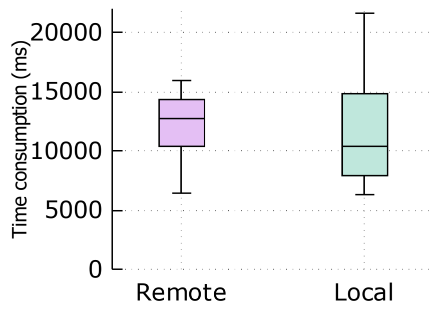


Fig. 2.4: Performance and energy consumption comparison of the Internet sharing case study



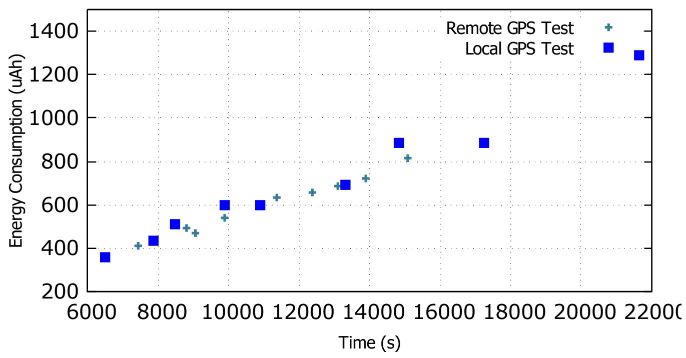


Fig. 2.5: Experimental results comparison between remote and local GPS access.

Experimental Setup: To set up testing environment without any modifications, we used Android x86 platform [2] on a powerful PC that was considered another peer having infinite battery capacities. Then, we emulated various network conditions using *Network Emulator for Windows Toolkit* [5]. For the comparison of our approach and cloud offloading, we re-performed the image processing test with a same small-scale image. When increasing the latency between the PC and the server from 0 to 150ms, the time and energy consumption also steadily increased as we expected. Then, we compared the cloud offloading-based results with our peer-to-peer based results reported in the previous. We have the following observations:

- The cloud offloading in WAN environment (i.e., limited and intermittent latency150ms)

[5]Microsoft Research. Network Emulator for Windows Toolkit (NEWT) version 2.1, 2010.

has limited benefits in terms of performance and energy efficiency compared to the P2P-based offloading.

- The cloud offloading in WLAN environment (i.e., moderate latency50ms) is sometimes more beneficial than the P2P-based offloading in terms of energy efficiency but does not have performance benefits.
- The cloud offloading in LAN environment (i.e., favorable latency0ms) completely overwhelms the P2P-based offloading regardless of the number of peers.

As a result, we can argue that different offloading mechanisms should be selected in accordance with various network conditions (i.e., latency/bandwidth characteristics) and different jobs (i.e., computation-intensive vs. sensor-based executions) in a dynamic, adaptive way to achieve further optimization [42].

2.5.3 Discussion

Despite its benefits, one can argue that executing any class files on a mobile device is harmful in terms of privacy and security. Thus, we will make our run-time system configurable as a future research direction. The implementation will provide a configuration file that can be used to specify user preferences with respect to battery, privacy, security, etc. In particular, a configuration file contains a set of key/value pairs, with the keys of `battery_level`, `trusted_host`, and `privilege`, which will indicate the favorable battery level for requested job executions, trusted network or peer list, and local resource access rights (e.g., CPU, memory, local files, or sensors), respectively.

The results presented above are subject to internal and external validity threats. The internal validity is threatened by how the interactive subject applications were exercised. The performance and energy consumption of interaction applications depend on how the user chooses to use them. To minimize this threat, the benchmarked use cases were fixed to using the same media (i.e., picture file), location (i.e., GPS coordinates), and Internet URL. Another internal validity threat is the fashion in which we implemented the jobs (e.g., blur effect). To minimize this menace, we took advantage of built-in Android libraries to implement these jobs.

The external validity is threatened by the mechanism to measure energy consumption. We measured the physical consumed energy directly using a power monitoring tool, which does not isolate the energy consumption of the subject applications from the total energy consumption of the whole mobile device. Thus, the energy consumption of the subject applications is subject to background services. To minimize this threat, we uninstalled all the third-party applications and stopped unnecessary background services.

2.6 Related Work

The work presented here enhanced the other efforts that optimize mobile applications performance and energy efficiency via remote executions including peer to peer networking, code migration and computation offloading. In the category of wireless peer-to-peer networks, there have been several research efforts. Built on top of WiFi Direct, Rio [43] leverages I/O system devices to capture and share contents and resources between the existing applications running on different devices without any modification. GameOn [44] also built on the same network infrastructure to establish non-Internet connection between gamers within closed range networks like in public transportation. CAMEO [45], and GigaSight [46] are also the similar content sharing systems in closed range network architecture. However, these contributions are missing a novel mechanism to pick up some appropriate devices out of the device pool for a specific remote execution.

Another relevant work to our approach is one that migrates different code bases into a system. In particular, code migration can be used to update existing, legacy systems [47,48]. Similar to our approach, code migration mechanisms are mainly used to run code on different memory spaces (e.g., running C++ code on multi-core systems [49], running JavaScript code on a server [24, 50], object-level migration for distributed systems [51], thread-level migration through middleware [52]). These code migration mechanisms also have affected execution offloading techniques in the mobile computing area.

Finally, our work shares objectives and techniques with execution offloading approaches

[53–55]. These offloading mechanisms is a well-known mobile application optimization technique that makes it possible to execute the applications energy intensive functionality at a powerful cloud-based server, without draining the mobile devices battery. However, these approaches have limits in terms of network availability, deployment cost and mobility. In this paper, to overcome those limits, we proposed our middleware system generalizing these offloading mechanisms using two different distributed execution modelsclient/server and peer to peer communications. As a result, we enabled resource-limited devices to extend their hardware capacities through our approach.

2.7 Conclusion

In this chapter, we have presented a distributed execution infrastructure to increase the quality of service of mobile applications as well as extend hardware capacities to resources-limited mobile devices. Our approach enables a mobile application to outsource any functionality through a novel middleware system by leveraging both cloud infrastructures and nearby mobile devices. In addition, our approach makes possible to bring a new hardware feature (e.g., sensors) to existing mobile devices. We have evaluated our approach by reducing the execution time and the energy consumption of case study applications as well as allowing a mobile device to use a GPS from nearby mobile devices. These results indicate that our approach represents a promising direction in developing complex mobile applications for resource-limited mobile devices.

CHAPTER 3
RELIABLE AND EFFICIENT MOBILE EDGE COMPUTING IN HIGHLY DYNAMIC AND VOLATILE ENVIRONMENTS

[6] [1]

3.1 Abstract

By processing sensory data in the vicinity of its generation, edge computing reduces latency, improves responsiveness, and saves network bandwidth in data-intensive applications. However, existing edge computing solutions operate under the assumption that the edge infrastructure will comprise a set of pre-deployed, custom-configured computing devices, connected by a reliable local network.

Although edge computing has great potential to provision the necessary computational resources in highly dynamic and volatile environments, including disaster recovery scenes and wilderness expeditions, extant distributed system architectures in this domain are not resilient against partial failure, caused by network disconnections.

In this chapter, we present a novel edge computing system architecture that delivers failure-resistant and efficient applications by dynamically adapting to handle failures; if the edge server becomes unreachable, device clusters start executing the assigned tasks by communicating P2P, until the edge server becomes reachable again. Our experimental results with the reference implementation show high responsiveness and resilience in the face of partial failure. These results indicate that the presented solution can integrate the individual capacities of mobile devices into powerful edge clouds, providing efficient and reliable services for end-users in highly dynamic and volatile environments.

[1] Minh Le, Zheng Song, Eli Tilevich and Young-Woo Kwon @ FMEC 2017

3.2 Introduction

Edge computing exploits data locality by processing massive amounts of sensory data collected by IoT devices "at the edge of the network." IoT devices, mobile devices, and edge servers process the locally collected data and transmit only the processed results to the cloud [56]. In addition, edge servers function as gateways that coordinate the at-the-edge computation by assigning tasks to the available connected devices for execution [57]. Traditional edge computing setups operate under the assumption of having a stable network connection, both between the edge server and the cloud, as well as between the local devices and the edge server.

In this chapter, we consider edge computing environments that are *highly dynamic* and *volatile* [58,59]. These environments are characterized by intermittent network connectivity, device mobility, and the presence of partial failure. In other words, the network connections both within the edge cloud and to the Internet are unstable or even non-existent. Users carrying the mobile devices involved can move at will, thus potentially affecting their devices' reachability to the edge cloud. Any of the computing devices involved, including the edge servers, can crash or become unreachable at any time.

The technical solutions presented herein enable reliable and efficient mobile edge computing in highly dynamic and volatile environments, such as those exemplified by disaster recovery scenes, battlefields, or expeditions to the wilderness.

3.2.1 Motivating Example

As a concrete example of the problem domain, consider Figure 3.1. A team of first responders reports to the location of a recent disaster. Each responder is supported by a collection of personal devices, both mobile and wearable. These devices are heterogeneous, in the sense that they differ in terms of their hardware resources, operating systems, and platform versions. A recovery vehicle hosts an edge server that also provides a WiFi access point (AP). Assume that the edge server's processing power is vastly superior to those of the personal devices, the WiFi AP covers the entire disaster recovery area, and the Internet connection's cellular signal is intermittent or non-existent.

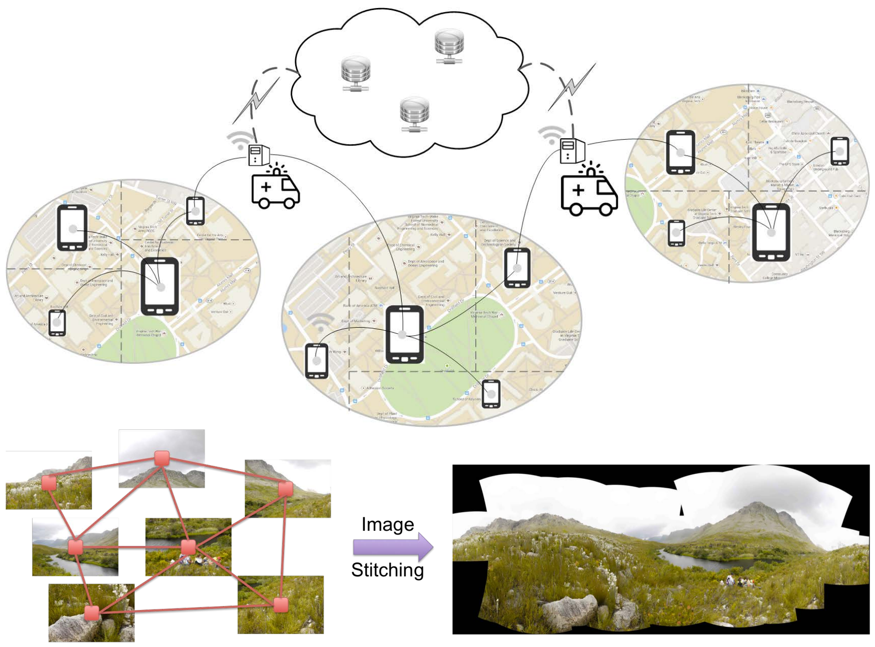


Fig. 3.1: Motivating example with image stitching service

Enabling the mobile devices and the edge server to cooperate as an edge cloud can greatly assist the responders in their mission. First, the responders must assess the situation on the ground and come up with a recovery strategy. To that end, they need to be able to efficiently map the entire recovery area. This task is called Image Stitching, a computer vision operation that glues adjacent pictures of an area together into a single panoramic image. The pictures are taken by the geographically dispersed responders using their respective devices, while the stitched panorama provides a detailed yet holistic view of the recovery area for the responders to facilitate their immediate recovery tasks (e.g., "What's around the corner from me?").

3.2.2 Technical Challenges

To realize the vision of enabling the heterogeneous mobile devices and the edge server to collaborate as a coordinated edge cloud, one must address two key technical challenges—device mobility and partial failure.

The responders need to move at will to attend to the recovery task at hand. This requirement entails that the devices forming the edge cloud may move to locations not covered by the limited WiFi AP, thus causing partial failure in the device-edge server distributed execution [60]. Therefore, another technical challenge is to provide not only an efficient and reliable edge cloud architecture, but also resistant to partial failure caused by device mobility [61,62]. The device-edge server distributed execution may operate in a peer-to-peer pattern, if the connection with the edge server is broken. The available network is likely to exhibit a high degree of volatility, with fluctuating bandwidth, latency, and packet loss rates [63,64].

For example, to stitch individual images into a single panoramic view, the individual devices that have captured their pictures need to send them to the edge server that has the computational and storage capacities to execute the stitching algorithm. However, the Image Retrieval service may operate in a peer-to-peer pattern, if the requested portion of the stitched map happens to be located on some nearby devices, or if the connection with the server is broken. The available network is likely to exhibit a high degree of volatility, with fluctuating bandwidth, latency, and packet loss rates [65].

This chapter describes a solution that provides reliable and efficient mobile edge computing in highly dynamic and volatile environments. Our solution comprises a novel service architecture that includes the service infrastructure, a trusted portable store of vetted mobile edge services, each of which constitutes a self-contained executable module to be downloaded to mobile devices and invoked at runtime. In addition, the coordination of mobile services and the edge server is orchestrated by our edge server architecture, in a context-adaptive, failure-resistant fashion.

This chapter makes the following contributions:

- **Mobile Edge Service Infrastructure**—a novel system component for deploying microservices on demand to heterogeneous mobile devices at the edge.

- **Adaptive Edge Service Architecture**—a novel distributed system architecture that dynamically re-configures itself to provide resilience to partial failure.

- **Empirical Evaluation**—We rigorously evaluate the effectiveness and performance/energy efficiency of our reference implementation on a series of micro and macro benchmarks and applications.

3.3 Technical Approach

In this section, we present our novel mobile edge service architecture, which enables reliability and efficiency in the face of the following technical challenge: the possibility of the mobile devices involved moving outside of their reachability range or failing for other reasons (i.e., partial failure).

To solve the aforementioned technical challenge, the architecture organizes mobile devices and edge servers into a hierarchical structure. All computing nodes, including the edge server and the set of available mobile devices, are connected using peer-to-peer communication interfaces, the main and the backup ones. When partial failure renders the edge server inaccessible, the backup interface makes it possible for the mobile devices to communicate with each other directly, with a cluster of mobile devices providing the edge services.

3.3.1 System Architecture

Here we provide an overview of our system architecture; the specific technical details are covered in the subsequent subsections, to which we provide forward references.

Our solution comprises a *service infrastructure for reliable and efficient mobile edge computing*. To understand how one can develop distributed mobile applications using this architecture, let us revisit the motivating example above. In that example, computing vision operations will be implemented as mobile microservice. Each service is a self-encapsulated unit of functionality managed by a service infrastructure. Each service includes a set of execution constraints that define the type of an edge computing device that can execute it in a given environment.

As the responders move around over time, the network topology of the mobile devices they are carrying is continuously changing. The technical challenge here is to hide the required underlying re-configurations of the mobile networks in response to the deployment of services on the continuously fluctuating—both in size and location—collection of mobile devices. We address these problems by providing *middleware support for dynamic and volatile environments*, whose adaptive facilities dynamically restructure the patterns of distributed communication in response to partial failure. Section 3.3.3 discusses the general design of the middleware system, while Section 3.3.3 discusses the details of handling partial failure.

3.3.2 Mobile Edge Services

Our service infrastructure features of application markets and mobile service repositories. Following the application market model enables mobile devices to automatically install and execute the required mobile edge services, while following the service repositories model enables mobile application developers to implement the required functionalities as service invocations.

Since the platform on which a mobile edge service will be executed is unknown until the runtime, service developers are expected to provide several equivalent versions for each service to support execution on all major platforms. An important design assumption is that of mobile devices possessing limited resources, with some of the limitations making it impossible for a given device to execute a given service.

3.3.3 Adaptive Edge Service for Reliability and Efficiency

Our middleware provides efficient communication support for mobile microservices, coordinating their executions between heterogeneous edge computing participants, such as the edge server and mobile devices. In addition to the communication support, due to the dynamic and volatile nature of wireless networks, the middleware provides a novel failure handling mechanism, activated in cases of service execution failures or network disconnections.

Figure 3.2 gives an overview of the system architecture. The primary service execution model is client-server executing microservices at the edge server, as long as it is reachable. However, when the network or edge server becomes unavailable, mobile services are executed by means of nearby mobile devices in a peer-to-peer model. Then, as soon as the network

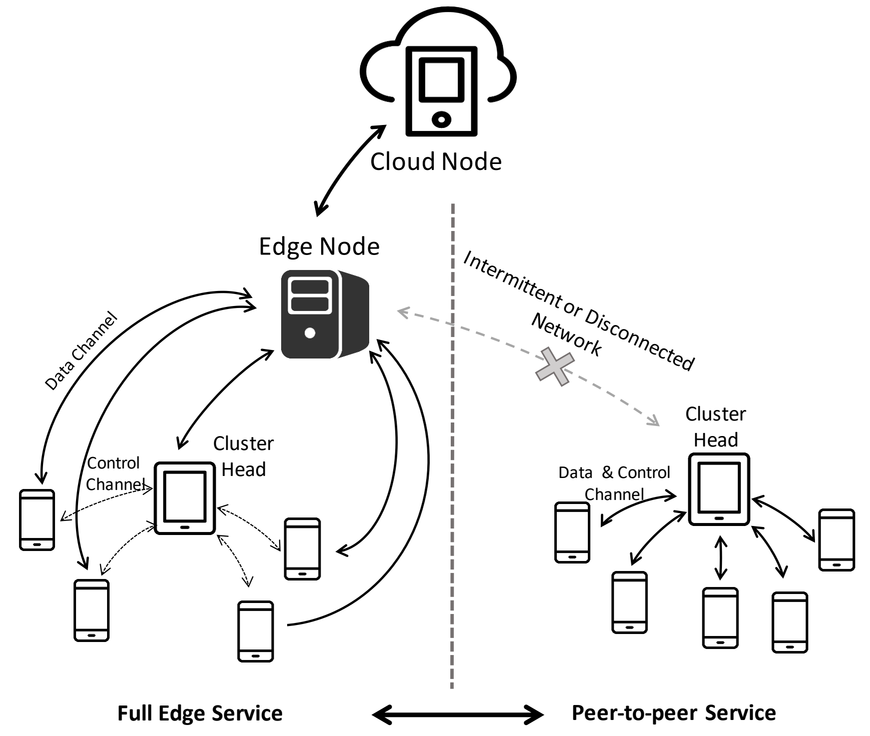


Fig. 3.2: Edge cloud architecture.

connection between the edge server and mobile devices is restored, the service execution model is switched back to the client-server model.

Service Execution

Although our system architecture consists of two communication models: the client-server model and the peer-to-peer model, we use an adaptive system architecture on top of the *topic-based* publish/subscribe middleware [66] with three main constituent components: *Broker*, *Worker* and *Client*.

The *broker* hosted on either an edge server or mobile devices plays a critical role in executing mobile services, including (1) receiving service execution requests from clients; (2) looking up and downloading execution packages from a service repository; (3) selecting workers based on their resource availability and capacity; (4) delivering service execution packages to the selected workers; (5) gathering execution results from the workers and returning them back to the clients.

The *worker* located on an edge server and all mobile devices reports its resource availability and capacity to the broker that assigns tasks by sending service packages to workers and then execution results are sent back to the broker.

The *client* requests mobile service execution to the broker and waits for the result from the broker. If the broker cannot handle incoming new service requests due to the limited number of available workers, the client immediately cancels the service execution request and executes it locally.

When the edge server is available and the communication model is the client-server model, the edge server functions both as the broker and the worker. When the edge server is not available, the mobile device which is the cluster head works as the broker, and the other mobile devices in the cluster are taken as available workers. For a client device, it ignores the differences in the network communication model, and only needs to coordinate with the available broker. If partial failure happens (e.g., no available workers), the broker returns "execution error" to the client device, and the client device executes the service locally.

Optimal Device Selection

The result of executing a service remotely is often dominated by the hardware configuration of the service execution device, as well as its network conditions such as bandwidth/delay characteristics [42]. Thus, our approach finds an appropriate number of devices for the current service execution environment to provide the best execution results with respect to performance. In the following discussion, we describe our service request algorithm in detail.

Our service request algorithm determines the best service execution model between an edge server and peers, and finds the most favorable devices based on the service execution capacity of each device. For the service execution capacity, we define the Device Responsiveness Level (DRL) metric, which is expressed as follows: $DRL = \sum_{i=\overline{1,M}} C_i \times C_{R_i}$, where

Algorithm 1 Service request algorithm.

```
1: function SELECTDEVICES($client$)
2:     workerList ← getAvailableWorkers()                              ▷ Include worker on edge
3:     if (workerList.edgeAvailable()) then                            ▷ Edge server is used
4:         $client$.offloadToEdge()
5:     else                                                            ▷ P2P is used
6:         $broker$ ← $client$.offloadToP2P()
7:         sortedDRLs[] ← sort(workerList.DRLs)                        ▷ Descending order
8:         maxCombine ← 0
9:         $availWorkerList$[] ← null
10:        for all ($DRL_i \in$ sortedDRLs) do                         ▷ Find max (i $\times DRL_i$)
11:            if (maxCombine < ($i \times DRL_i$)) then
12:                $availWorkerList \xleftarrow{\text{add}}$ workerList[i]
13:                maxCombine ← ($i \times DRL_i$)
14:            end if
15:        end for
16:        if (maxCombine < $DRL_{client}$) then                        ▷ Prefer local execution
17:            $broker$.runAtLocal($client$)
18:        else                                    ▷ Job is distributed to the selected workers
19:            $broker$.sendToWorkers($availWorkerList$)
20:        end if
21:    end if
22: end function
```

C_i is the CPU speed of core i ($C_i = CPU_i$); C_{R_i} is the remaining percentage of the CPU core; and M is the number of cores. To find the most favorable mobile devices in a P2P network, a `Broker` selects N devices by comparing their DRL values with the client's DRL, divides a job into N equal pieces[2], and sends them to the corresponding workers through the following steps:

- Assume that we have D devices. First, we sort the available devices based on their processing power (DRL) in the decreasing order. Here, we have a list of devices $\mathcal{D} = \{d = 1, 2, ...D\}$, with their DRL $\{DRL_d \mid \forall d \in \mathcal{D}\}$, $DRL_{d1} \geq DRL_{d2}, \forall d_1 < d_2 \in \mathcal{D}$.
- For each $d \in \mathcal{D}$, we calculate the $d \times DRL_d$, and select $N = \max(d \times DRL_d), \forall d \in \mathcal{D}$. Then, select all devices from the sorted list that has higher DRL values than d, and the selected device set is: $\mathcal{N}^* = \{1, 2, ..., N\}$. The basic intuition behind this arrangement is, considering that the job will be divided into N equal pieces, the overall execution time can be estimated as the longest execution time of executing one piece on each

[2]To make the data partitioning problem simple, we adopted the Hadoop's idea that splits data into multiple chunks of the same size.

worker, or say, the execution time on a worker device with the lowest DRL of all selected workers. If we use I to represent the job, and I/N to represent the piece on a worker, the overall execution time can be represented as $t = \frac{I}{N \min(DRL_d)} (\forall d \in \mathcal{N})$. As we sort the devices based on their DRL in decreasing order, $t = \frac{I}{N \times DRL_N}$, for the Nth device has the lowest DRL on all selected devices. Therefore, to minimize the overall execution time, is to first find a device d to maximize $d \times DRL_d$, and select all the devices whose DRL is higher or equal to d.

- We further consider the DRL of the client. If $DRL_{client} \geq N \times DRL_N$, which means that executing the job on the client is faster than executing the job on the peers, we choose to execute the job on the client; Otherwise, if $DRL_{client} < N \times DRL_N$, we choose to distribute the job equally into N pieces, on the selected workers $\mathcal{N}^*$.

Handling Network and Service Failures

Clients maintain two communication channels with an edge server and a cluster head. By exchanging a heartbeat message between an edge server and clients, each client can check the availability of the edge server and network status to both the edge server and nearby devices. If a client does not receive an acknowledgement from the edge server due to the network disconnection, it informs nearby clients of the network failure through a cluster head. Then, clients immediately switch their service execution model to the P2P mode and continue the failed service execution through a peer-to-peer network.

Algorithm 2 explains our failure handling strategy. Any clients maintain one heartbeat communication to the edge server and they periodically send heartbeat requests to the server and wait for the response. If the response is not received within a timeout, a `failedEvent` will be dispatched to the holder to notify of the network failure. Once the client receives `failedEvent`, it immediately switches its service execution model to the P2P mode and continues the failed service execution. By exchanging heartbeat messages with a the cluster head, the client can also detect a service failure in the P2P mode and execute the failed service locally. In the meantime, the client keeps attempting re-connection to the edge

Algorithm 2 Handling network and service failures.

```
1: function CHECKHEARTBEAT()
2:     openNewHBConn()                              ▷ open new Heart-beat connection
3:     while (Thread.isInterrupted()) do
4:         try
5:             wait(HEARTBEAT_PULSE)                ▷ HEARTBEAT_PULSE = 3s
6:             sendHeartbeatToEdge()
7:             resp ← waitForResponse(TIMEOUT)      ▷ TIMEOUT = 2s
8:             if (resp != null) then
9:                 notifyOKEvent()
10:            else                                 ▷ If unable to receive response
11:                notifyFailedEvent()
12:            end if
13:        catch (NetworkException)
14:            try
15:                wait(REESTABLISH)                ▷ REESTABLISH = 3s
16:                closeCurrentHBConn()
17:                openNewHBConn()                  ▷ Reopen heartbeat connection
18:                notifyRestoredEvent()            ▷ Network has been restored
19:            catch (Exception)
20:                                                 ▷ When attempt failed again
21:                                                 ▷ Silently start a new loop
22:            end try
23:        end try
24:    end while
25:    closeCurrentHBConn()                         ▷ Close current Heart-beat connection
26: end function
```

server by sending a heartbeat request. If the client receives a heartbeat response from the edge server, it dispatches `restoredEvent` to the holder and restores the system to the normal state[3].

3.4 Evaluation

We evaluate the effectiveness of our approach through a micro benchmark and realistic case studies. Specifically, we conduct two test cases to ascertain how efficient and reliable our system would be in highly dynamic and volatile mobile edge computing environments[4]. The testbed for experiments has been built up with various Android devices featuring WiFi Direct and one edge server.

[3]For the details, please see our demo at: http://youtu.be/7dd1EQFb_vk

[4] Our system implementation and evaluations can be found at: https://github.com/minhld/Pub-Sub-Middleware

Table 3.1: List of devices used in the experiments in chapter 3.

Device	**CPU**	**RAM**	**Battery**	**OS**
Moto G4	Octa 1.5GHz	2GB	3000mAh	6.0.1
G4	Quad 1.5GHz	3GB	3000mAh	5.1
Asus ZF2	Quad 1.7GHz	3GB	2400mAh	5.1
BLU R1	Quad 1.3GHz	1GB	2500mAh	6.0
S3	Quad 1.4GHz	1GB	2100mAh	4.4
Dell PC	i7 3.6GHz	8GB	N/A	Win10

3.4.1 System overhead

In this experiment, we evaluate the system's overhead by measuring the execution time of each main component of our system architecture, which include the client, the broker, and the worker, both on the edge server-based and the peer-to-peer-based networks. We (1) pre-install an empty service, which sleeps for 15 seconds on the edge server and P2P workers and (2) place mobile devices in a nearby area for fast, stable WiFi Direct communication. We timestamp the start and end times of each component executing its job and then aggregate all the times, excluding the service execution time (i.e., 15 seconds). Figure 3.3 shows the total overhead time, which can be disregarded.

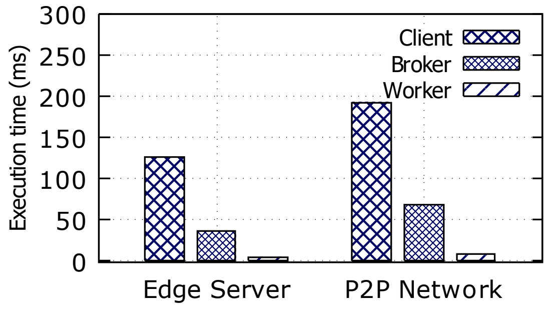


Fig. 3.3: Aggregate overheads in two different networks.

3.4.2 Case Study

To evaluate our system's implementation in realistic scenarios, we conduct two case studies:

- **Image Processing Service:** splits an image into several parts, and each device then blurs one image part and sends the results back to the client, which merges all these partial results into a single image.

- **Internet Sharing Service:** provides Internet connectivity to nearby devices that lack a cellular data plan. The service loads a web page in the device's mobile browser by engaging surrounding Internet-connected devices.
- **Word Counting Service:** splits an e-book into several text parts, and each device then find most frequent words and sends the results back to the client, which merges all these partial results and shows the top 50 most frequent words.

Image Processing Service

For this experiment, we implemented an image blurring service using the Gaussian convolution. An image is split into N equal parts vertically, and then each part is sent to a worker. Each worker executes the Gaussian convolution definition to blur its image part and sends the result back to be merged into a complete image.

We first requested the service to the edge server, and then disconnect the network between the edge server and a client, resulting in switching the service execution model to the P2P mode. We measured the total execution time of the client that elapsed between the initiation of the service request and the arrival of all results. Figure 3.4 shows that the edge server-based and P2P-based service execution. Although the edge server-based service execution outperforms the performance of P2P-based service executions, as the number of mobile devices increases, the performance of the P2P-based service model is also significantly improved.

Then, to evaluate our peer selection approach, we compared the estimated execution time with the actual time taken by the P2P collaboration in five scenarios, which engage between 1 and 5 devices. Figure 3.5 (Top) shows the estimated and actual lines having the same trend and close values. We found that this trend approximation also occurs when starting the service execution from different devices in the P2P network. Figure 3.5 (Bottom) describes the similar trends on 3 devices with different resource capacities, when requesting the same service from different mobile device in the P2P network.

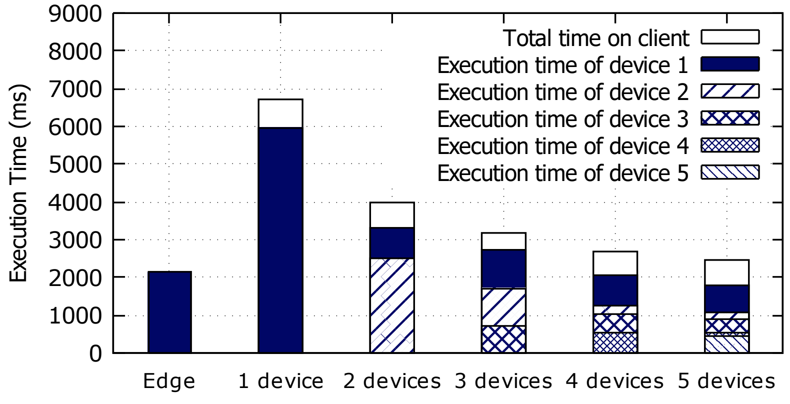


Fig. 3.4: The performance comparison of the edge server- and P2P-based execution model in image processing test.

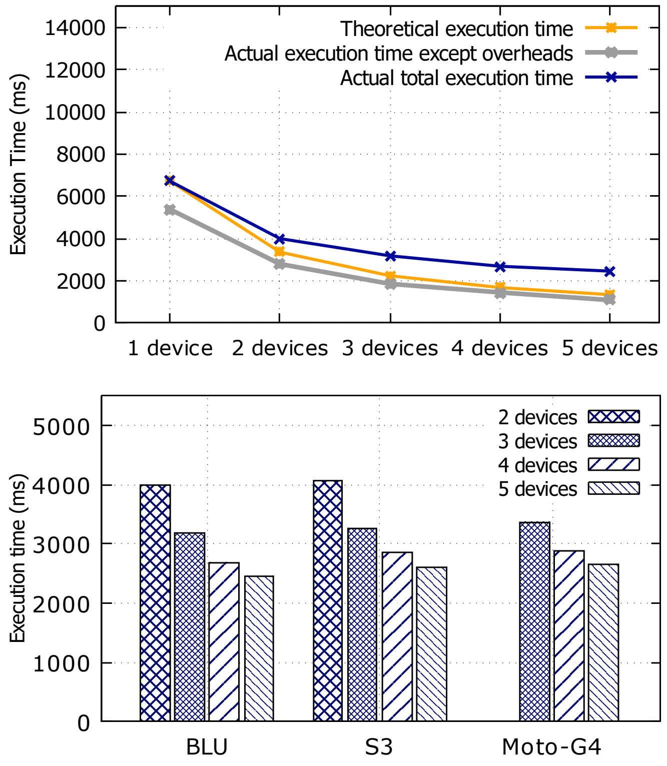


Fig. 3.5: Comparison of the estimated versus actual execution times in image processing test.

Word Counting Service

This service will be used mostly in the section 3.4.2. Although a word-counting service would not represent a typical example targeted by our solution, we use this canonical computation-intensive scenario to compare the respective performance and energy efficiency of the edge server and the P2P networks under multiple configurations. Figure 3.6 shows the similar trends as in the image processing service above, indicating that our scheduling algorithm allocates the optimal number of devices to maximize the performance and minimize the energy consumption.

Figure 3.6 shows the similar trends as in the image processing service above, indicating that our scheduling algorithm allocates the optimal number of devices to maximize the performance and minimize the energy consumption.

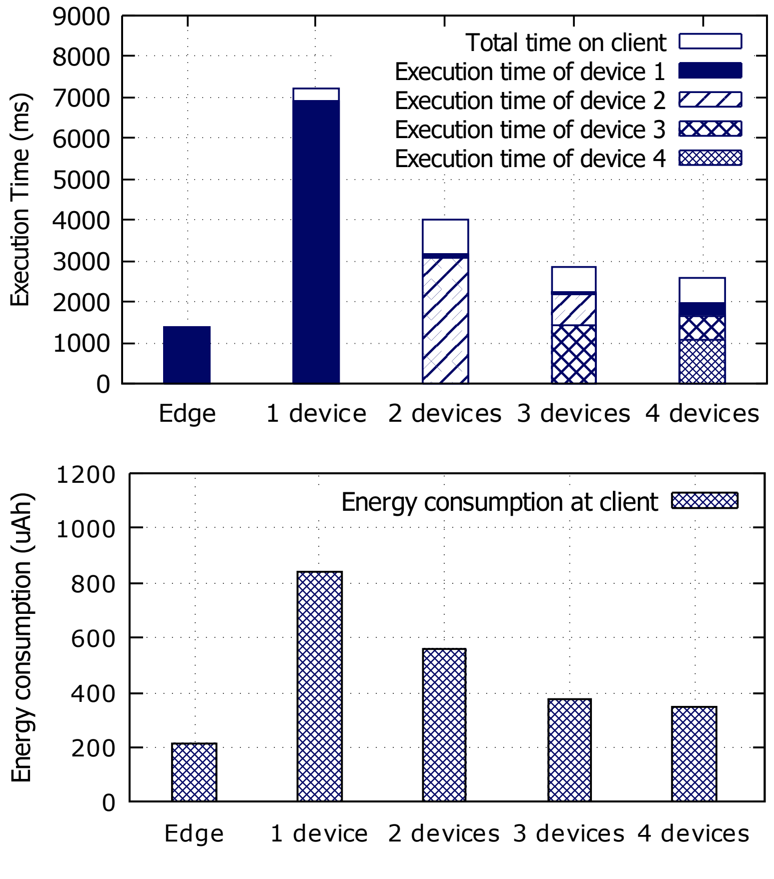


Fig. 3.6: Word-counting: execution time and energy consumption.

Figure 3.7 evaluates the accuracy of Algorithm 1 by comparing its output with the actual measured performance numbers, summarized in Figure 3.6. The graph shows that the two lines follow the same trend, indicating that the algorithm approximates the actual performance with an acceptable level of accuracy.

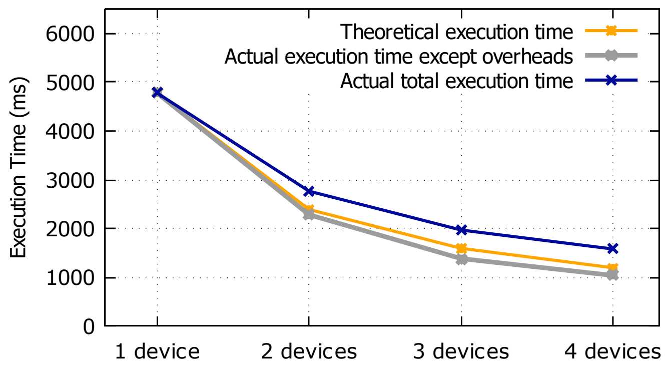


Fig. 3.7: Comparison of the estimated versus actual execution times in word counter test.

Internet Sharing Service

Next, we study the Internet sharing service, an example of sharing the nearby devices' computing resources, including networks, files, sensors (e.g., GPS, motion, environmental etc.). We design the Internet sharing service to distribute requests including URL, n – the number of devices in the cluster, and $index$ – the index of a device. Each device (including the client) downloads the HTML text contents of a web-page from the URL, collects its resource URLs (images, audio, videos etc.) and downloads a batch of URLs according to its $index$ position of n devices.

In this experiment, we assume that only one device, lacking Internet access, sends the same content-downloading request to an edge server or a cluster head. The first two bars in Figure 3.8 show the total execution time measured at different devices when connecting to the edge server, while the next five bars show the total execution time on each mobile device when connecting to different cluster heads. To show how dissimilar resource capacities lead to different performance characteristics, we configured our testbed to sequentially select different devices as the cluster head for each experiment. Unlike the first two bars, which are almost the same, the cluster heads have dissimilar resource capacities, with their respective performance levels fluctuating in the wider range between 37 and 42 seconds, being obviously slower than the edge server.

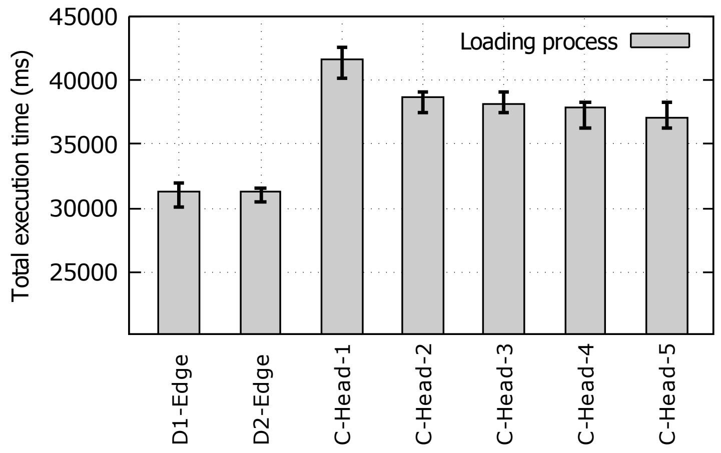


Fig. 3.8: The edge server versus P2P-based service execution in internet share test.

Failure handling

As discussed in Section 3.3.3, our failure handling mechanism can switch the remote execution back and forth between the edge server and the P2P network in response to the edge server becoming unavailable and available again, as well as the network getting disconnected and reconnected. We evaluate the efficiency and effectiveness of our failure handling mechanism by implementing a word counting service, a well-known use case of multi-processing setups, such as Apache Hadoop.

The word counting service searches for 50 most frequently used words from a textbook. First, a client loads the entire text from a book and attaches it to the request message. Then, a broker finds the number of available workers, divides the text into the same number of parts, and sends them to the workers.

Switching to a P2P network

First, we enable the edge server, so that the client can offload the word-count service normally there within 4.3 seconds (Figure 3.9). Then we repeat the test but shut the edge server down at the 3rd second; then the client detects the system failure at around the 5th second (the timeout for failure detection is 3 seconds) and immediately initiates a new offloading session to the P2P network. The P2P collaboration returns the result within 23 seconds, and the total process takes around 29 seconds. From this moment, if we offload the service again, the offloading will be dispatched directly to the P2P and completed within around 23 seconds.

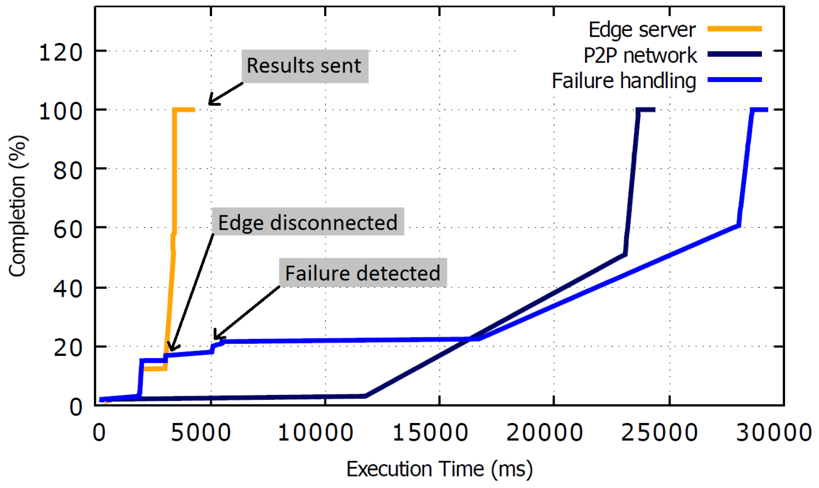


Fig. 3.9: Execution time of the word count service measured in 3 different cases.

Edge server restoration

In this experiment, we observe and examine how our system behaves in consecutive scenarios: (1) run normally on the edge server, (2) detect failure, (3) switch to P2P, and (4) restore back to the edge when the network reappears. To this end, we use the same test-bed as in the above experiments, thus enabling the client to offload a number of service requests; then we measure the total time of each request at the client. In this test, the client delivers three different services: empty, word-count, and image processing (the image processing service is discussed in Section 3.4.2).

Figure 3.10 depicts the overall performance of our system in each type of service requests with a number of continuous attempts. Particularly, in the case of word-count service, the

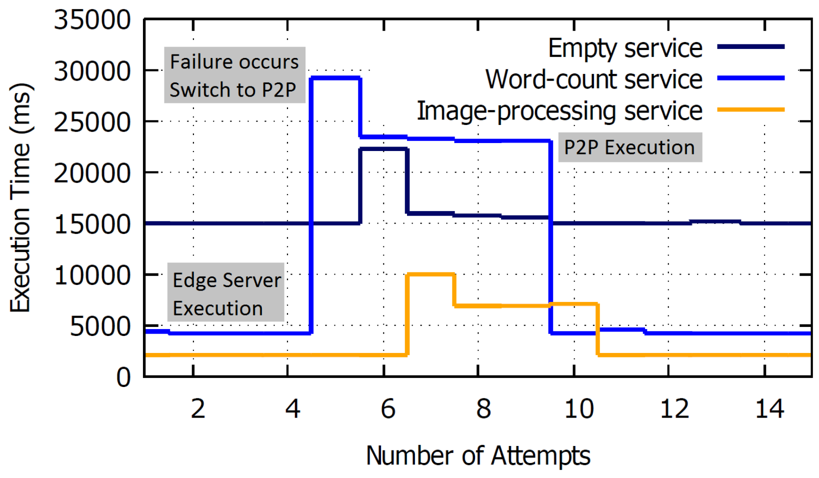


Fig. 3.10: System performance when requesting multiple services.

first 4 requests are resolved by the edge server within 4.3 seconds. During the 5th request, the edge server is shut down, with the client immediately detecting this event and switching to the P2P execution mode, completing the request in 29 seconds. After that, the upcoming requests from the 6th to 9th requests are resolved by the P2P configuration in 23 seconds on average. When the edge server comes back, the client reestablishes the connection and pushes the 10th and subsequent requests to the edge, thus reverting to the original normal execution mode.

3.4.3 Discussion

Our approach offers a number of advantages and has several limitations that we discuss in turn next.

Advantages

Mobile edge services serve as building blocks, enabling application developers to design mobile applications that take advantage of the complimenting strengths of nearby devices from the resource provisioning perspective. In other words, developers can orchestrate the execution of an application's functionalities on the devices whose resources are the most suitable and abundant for given execution tasks.

This software architecture eliminates the need to accept executable code from nearby devices by introducing a trusted third-party intermediary. The introduction of the intermediary component increases the trustworthiness of the distributed execution model. However, for this intermediary component to become widely applicable, multiple stakeholders in the technology will need to come to an agreement, which may be hard to accomplish.

Once a service is downloaded to a mobile device, the Internet connection is no longer required, making it possible to execute mobile services in environments with limited or intermittent wide-area networks. In fact, this architecture can increase the utilization of nearby mobile devices in such execution environments.

Finally, the ability to address resource scarcity makes this architecture potentially suitable as a solution for orchestrating the execution of IoT setups, in which each participating device is known to posses specialized unique functionality (e.g., sensing, media capture, etc.) while lacking general hardware or software resources.

Limitations

The trustworthiness of mobile edge services hinges entirely on the reputation of the trusted intermediary component, thus restricting this distributed execution model to environments that provide such trusted components. In addition, the high dynamicity of the mobile context increases the risk that no suitable nearby device may end up being available for executing a service with a specified set of requirements. To defend against these risks, mobile developers need to provide back-up options for executing services, either with relaxed QoS requirements or using the local resources.

Since one cannot predict what platforms will be run by the available nearby devices, service developers have no choice but to provide multiple versions of their services, equipped to run on all major platforms. This design feature increases the developer workload, even though JavaScript execution may provide a reasonable cross-platform solution.

3.5 Related Work

The work presented here is related to other complementary efforts that improve the reliability and efficiency of mobile distributed execution, including frameworks, peer-to-peer networking, code migration, and computation offloading.

Alljoyn[5] is an open source framework that hides the complexity of network communications for application programmers. By providing interoperability between multiple platforms without any transport layers, Alljoyn makes the integration and initiation of network communication easy and straightforward. Before the Wi-Fi Direct, many efforts have focused on optimizing peer-to-peer network based on existing short-range/wireless communication technologies available on mobile devices including Bluetooth, Wireless IEEE802.11 and cellular communication link [67, 68].

In the category of wireless-based P2P communication, before the Wi-Fi Direct technology, several efforts utilized other wireless communications to establish P2P networks, such as media sharing system sin urban transport using Bluetooth [69], resource sharing using cellular networks [70], and radio resource sharing over ultra-wideband [71]. Built on top of Wi-Fi P2P, Rio [43] leverages I/O system devices to capture and share contents and resources between the existing applications running on different devices without any modification. Some of their applications are multi-system photography and gaming, singular SIM card for multi-devices, music and video sharing.

Another related work direction is code migration to update existing, legacy systems [72]. Similar to our approach, code migration mechanisms are mainly used to run code in different memory spaces (e.g., running C++ code on multi-core systems [73], running JavaScript code on a server [24], object-level migration for distributed systems [51], thread-level migration through middleware [52]). These code migration approaches have influenced the design of offloading mechanisms in the mobile computing area.

Finally, our work shares objectives and techniques with execution offloading [3,74], well-known mobile application optimizations that execute the resource-intensive functionality at cloud-based servers to avoid draining the mobile device's battery. In this paper, we generalize these offloading mechanisms using two different distributed execution models—client/server and peer-to-peer communication models.

[5]https://allseenalliance.org/framework

3.6 Future Work and Conclusion

As a future work, we plan to explore additional diverse failure handling mechanisms by exposing them as reusable components, activated in response to the underlying system behaving abnormally, based on our prior work on hardening remote services with network volatility resilience [75].

In this paper, we present a service middleware architecture and reference implementation that execute services reliably and efficiently on available devices, both mobile and stationary, accessed via self-adaptive communication channels. Our solution centers around the characteristics of highly dynamic and volatile environments, in which the network connection between the devices forming the edge cloud is intermittent. The presented solution automatically detects and handles partial failure of the network, by switching between client-server and peer-to-peer mobile edge execution modes. Our experimental evaluation shows that our solution enables resilient and efficient mobile edge execution over unreliable networks, typical of highly dynamic and volatile environments, heretofore not supported by edge clouds.

3.7 Acknowledgment

This research is supported in part by the National Science Foundation through the Grant CCF-1649583.

CHAPTER 4 ENABLING FLEXIBLE AND EFFICIENT REMOTE EXECUTION IN OPPORTUNISTIC NETWORKS THROUGH MESSAGE-ORIENTED MIDDLEWARE

[7] [1]

4.1 Abstract

Computation offloading has received much attention to improve the performance or energy efficiency of mobile systems that have usually limited and constrained resource capacities. Yet, applying the computation offloading technique in opportunistic networks that are highly dynamic and often become volatile still remains a challenge due to the following reasons: (1) technical difficulties in constructing efficient and reliable execution environment using commodity devices using WiFi, (2) a lack of runtime support for multiple clients that request diverse computational tasks and execute them concurrently with minimum performance impacts. In this chapter, we introduce a new middleware system that provides an offloading framework operated in opportunistic networks. In particular, our middleware employs a publish-subscribe communication mechanism to provide multiple different communication models (e.g., one-to-one, one-to-many, many-to-one, and many-to-many) for different use cases. Furthermore, when distributing computational tasks to nearby nodes, our middleware takes their resource capabilities into consideration for efficient execution. Finally, since partial failure is an unavoidable artifact in highly dynamic and volatile opportunistic networks, we provide a simple, but effective failure handling mechanism. Our benchmarks and experimental results indicate that our approach enables programmers to easily apply computation offloading techniques in opportunistic networks when compared with the local execution.

[1]Minh Le, Young-Woo Kwon @ COMPSAC 2017

4.2 Introduction

Despite the significant development of mobile devices, the resource demands of mobile applications often outstrip the hardware capacities of mobile devices. One approach to address the resource constraint problem is *computation offloading*—executing CPU-intensive functionality at a powerful cloud-based server, thereby improving both performance and energy efficiency [4, 74, 76]. While the computation offloading technique has been well studied in the field of mobile cloud computing in which powerful servers and fast, stable network connection are mostly available, applying them in opportunistic network that are highly dynamic and often become volatile still remains a challenge.

Recently, the WiFi Direct (WFD) technology has received much attention in distributed systems and mobile computing utilizing opportunistic networks in closed ranges where Internet is unnecessary [43, 44]. The WFD technology allows a mobile device to discover nearby ones and establish connections between them using an ordinary WiFi protocol without a WiFi infrastructure such as an access point (AP). However, the current studies are mostly limited to data distribution among clients [44]. Furthermore, due to the limitation that a wireless network organized through WFD only provides 1-to-1 and 1-to-N communications, despite its convenience it has been less utilized in *computation offloading* or *remote execution*. In particular, multiple connections between devices is generally unsupported by default [29]. As a result, to achieve flexible and efficient communications between devices in a WFD network (i.e., P2P network), a programmer must deal with complex network-related programming.

In this chapter, we address the following research problems to enable *computation offloading* in a small, volatile wireless network with no necessity of powerful computing resources.

- How can one efficiently offload its software functionality to nearby mobile devices without changing network configurations?
- How can the participants of a P2P newtork deal with network or service failures?

- How can programmers easily construct a peer-to-peer network between nearby devices for their applications?

To that end, we first provide a middleware system that can construct different types of networks using the WFD technology. Then, we build our computation offloading infrastructure on top of the newly developed middleware system. To support multiple message delivery models, we extend message-oriented middleware (MOM) that can construct a flexible and reliable peer-to-peer (P2P) network using the WFD. In particular, we employed a broker from traditional publish-subscribe middleware, so that any mobile device in a P2P network can initiate communications through a broker (i.e., any mobile device can offload a job without reforming a P2P network at the same time). The broker monitors the resource usages of all network participants and manages job request/response queues on each device for job distribution. To fairly distribute computation units (i.e., job in this chapter) to P2P network participants, our approach takes their hardware resource capacities into consideration to determine the appropriate amount of job. Moreover, our middleware implementation can be used as a general-purpose communication middleware. Finally, due to the volatile and dynamic characteristics of wireless networks, any mobile device may leave a peer-to-peer network at any time; a network can be destroyed; and transmitted data can be damaged or lost. Thus, we provide a simple recovery mechanism for partial failures. This chapter makes the following contributions:

- **Flexible and reliable communication infrastructure:** The newly developed middleware allows programmers to easily develop an application that requires various communication mechanisms in a volatile opportunistic network.
- **Efficient task allocation algorithm:** We introduce a resource-based task allocation algorithm to optimize the execution of offloaded tasks and reduce their impacts on mobile devices.
- **Empirical evaluation:** We evaluate the effectiveness and efficiency of our approach through a series of benchmarks and case study applications.

The rest of this chapter is organized as follows. Section 4.3 describes the technologies we used. Section 4.4 presents our approach. Section 4.5 empirically evaluates our approach. Section 4.6 compares our approach and other closely related approaches, then we conclude our work in Sections 4.7.

4.3 Background

WiFi Direct (WFD)

The WiFi Direct technology is a new peer-to-peer standard built on top of the IEEE 802.11 to provide direct connections between Wi-Fi devices without an Internet connection [77]. Over a Wi-Fi network, a device can discover and connect to other devices of any type without special configurations or setups. Once a connection is established, the devices can communicate with each other as a Client or a Group Owner. Wi-Fi Direct has been widely used to transfer content or share applications between the nearby mobile devices. Our approach uses Android WiFi Direct[2] to utilize nearby computing resources without an Internet connection or a wireless router.

Message-oriented Middleware (MOM)

In distributed computing, MOM has been widely discussed [78] and there are various implementations for cloud or mobile systems such as ZeroMQ [79], RabbitMQ [66] or ActiveMQ [80]. Since MOM supports multiple messaging paradigms including *point-to-point*, *fan-out*, *publish/subscribe*, *request/response*, its implementations have been used for various systems, especially for server applications. Offloading a local execution unit to nearby devices in an opportunistic network requires different execution models for different application scenarios or network constructions. As a result, to provide a flexible and reliable offloading infrastructure, we adopted message-oriented middleware, so that a programmer

[2]Android: WiFi Peer-to-Peer, `http://developer.android.com/guide/topics/connectivity/wifip2p.html`

can choose an appropriate messaging paradigm for different applications. As a reference implementation, we extended ZeroMQ[3].

4.4 Approach

In this section, we present our middleware system that supports reliable and efficient computation offloading and message delivery in a volatile wireless network.

4.4.1 Architecture Overview

In opportunistic networks relying on WFD or Bluetooth, multiple connections among devices is generally unsupported by default [29], thereby making it impossible to initiate computational offloading requests from different devices without changing a network topology [1]. Specifically, a group owner can only send a message to all group participants (i.e., 1-to-n communication), while group participants cannot exchange messages among themselves (i.e., n-to-n communication). As a result, a mobile device needs to re-construct a new peer-to-peer network for every single communication. Thus, from our prior work [5], we identified the following technical challenges to achieve efficient and reliable computation offloading in a peer-to-peer network: (1) offloading jobs or delivering messages among nearby mobile devices without changing their network configurations and (2) handling network or service failures in intermittent or disconnected networks.

To that end, we designed a *topic-based* publish-subscribe middleware that consists of the following components: *Broker*, *Worker* and *Requester*. Figure 4.1 shows the system architecture of our middleware system. In the following discussion, we explain our middleware system with a focus on computation offloading, and then discuss how our approach can be generalized for other application development. First, *Requester* (which is a consumer in a typical pub-sub pattern) sends a job request message to the *Broker* which then divides the request into smaller tasks and disseminates each to one *Worker* (or publisher in pub-sub) to execute. The Worker executes task and returns a result back to the Broker.

[3]We have experimented with RabbitMQ and ZeroMQ for our reference implementation and then selected ZeroMQ because of the performance and the ease of modification.

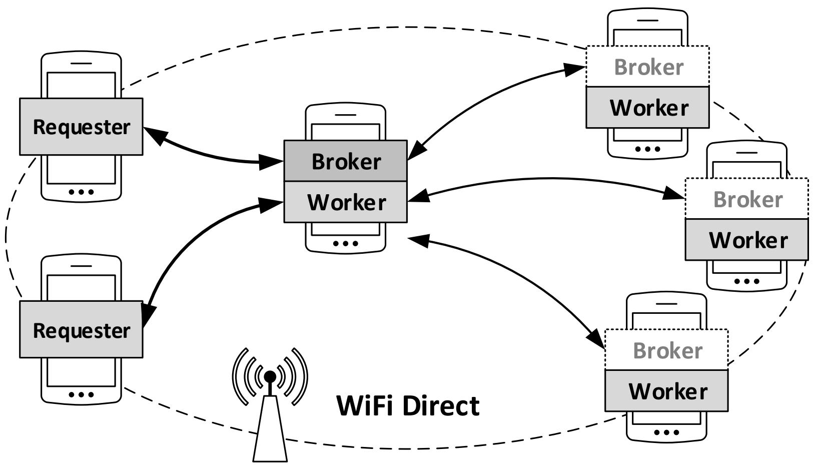


Fig. 4.1: System architecture.

To send a complete result back to the Requester, the Broker merges all the partial results received from the assigned Workers. In this workflow, the Requesters can start sending job requests simultaneously at any time, and then the Broker schedules job executions as soon as Workers are available. Figure 4.2 describes the details of the aforementioned workflow. In addition, because the three constituent components are installed on every device, they can dynamically switch their roles in accordance with the need for network topology changes (e.g., in case of network failures).

4.4.2 Flexible and Efficient Group Organization

By the decentralized design, our middleware can be deployed as various combinations of components to form a different network topology. For example, a device can host two Workers running on different threads, or both Requester and Worker to easily switch its role. To that end, we provide two group models—brokerless and broker-enabled model. In the following discussion, we describe each model in turn.

Brokerless group

The bokerless group (i.e., 1-N group, Figure 4.3-left) comprises of all three components, but a Requester and a Broker are installed on the same Group Owner (GO) device. Because in a WFD network, one client cannot establish multiple connections to different GOs directly

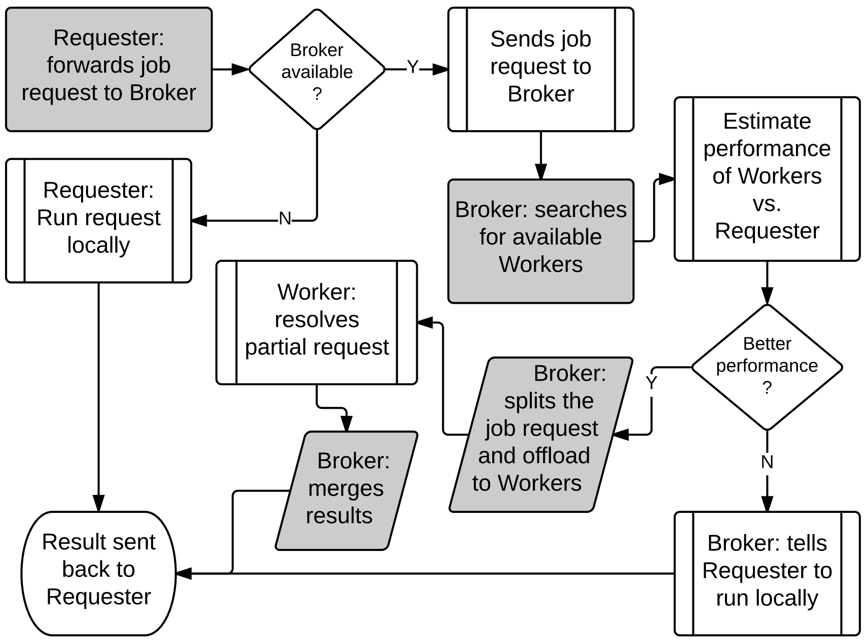

Requester: forwards job request to Broker
Broker available ?
Y
N
Sends job request to Broker
Requester: Run request locally
Broker: searches for available Workers
Estimate performance of Workers vs. Requester
Worker: resolves partial request
Broker: splits the job request and offload to Workers
Better performance ?
Y
N
Broker: merges results
Result sent back to Requester
Broker: tells Requester to run locally

Fig. 4.2: System detailed workflow

at the same time [13, 29], the Brokerless group is only useful when a network doesn't need to change; an application needs to be run fast; a programmer wants to simply configure *point-to-point*, *fan-out* and *request-response* patterns.

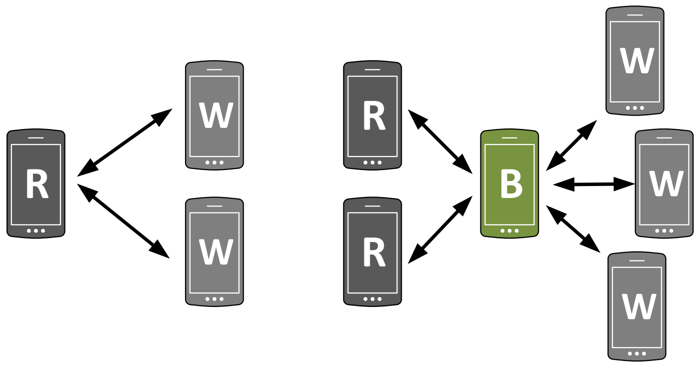


Fig. 4.3: Brokerless and Broker-enabled models.

Broker-enabled group

The broker-enabled group (i.e., N-N group, Figure 4.3-right) requires a Broker to be installed on the GO device. This model enables flexible network topology, where multiple Requesters and Workers are able to easily join or leave a group at run-time as well as exchange messages in a parallel fashion. As a result, a programmer can easily configure *request-response* and *publish-subscribe* patterns in a peer-to-peer network. However, the broker-enabled group has the following limitations: (1) one device must be dedicated as the Broker as long as the group endures, (2) because only one Broker is allowed at a time, the Broker may become a performance bottleneck, (3) if the Broker goes down for any reason, a group needs to be re-constructed.

4.4.3 Remote Job Execution

For efficient execution, we introduced Device Responsiveness Level (RL) which is a value representing resource information of mobile device. It is directly proportional to the resource capacities (number of cores, CPU speed, total memory and battery) and their percentage of remaining. In our first implementation, we only consider CPU information as the main factor to the RL value and compute RL as follows:

$$RL = \sum_{i=\overline{1,N}} CPU_i \times RP_i \tag{4.1}$$

where N is the number of CPU cores, CPU_i is the clock speed of CPU i in GHz and RP_i is the remain percentage of CPU i. Obviously, the higher value of RL means the greater computation availability of the peer. The actual clock speed of each CPU in the Equation 4.1 can be simply estimated on worker device by extracting from `cpuinfo_max_freq` [4] system file; and the available percentage of each CPU can be inferred accordingly from `/proc/stat` file as follows:

$$RP_i = 1 - \frac{(\sum T_{Act_2} - T_{Idle_2}) - (\sum T_{Act_1} - T_{Idle_1})}{(\sum T_{Act_2} - \sum T_{Act_1})}$$

where $\sum T_{Act_x}$ and T_{Idle_x} $(x = 1, 2)$ are the total active time and idle time in two consecutive inquiries on device i.

[4]Located at: `/sys/devices/system/cpu/cpu0/cpufreq`

Job Partitioning

Next, for the efficient offloading, a Broker divides job data into N pieces which are proportional to RL values, so that each task has the size M_i as follows:

$$M_i = M \frac{RL_i}{\sum_{j=\overline{1,N}} RL_j} \tag{4.2}$$

where M is the total size of data in bytes, N is the number of available Workers, and j-device has its RL_j value respectively. Then, the Broker creates N tasks, each of which is a replica of the original message except for the partial data with M_i size and dispatches them to the Workers.

Support for Multi-Clients

To support multiple job requests from different clients, we provide a simple job scheduler on the Broker as follows: the Broker keeps two queues (1) a request queue holding incoming requests and (2) a result queue holding results returned from the Workers. The Broker periodically checks the request queue. If the queue is not empty, the Broker pops out the first item and collects status of the available Workers. If the remaining resources of the available Workers are not sufficient, the Broker postpones the corresponding request. Otherwise the request is processed. This procedure continues until there are no items left in the queue.

Each Worker also has the request and result queues to handle multiple task executions. When a task request is popped out from the queue, a Worker dedicates a separated thread to execute that task. When the task is successfully executed, the result is stored in the result queue and then sent back to the Broker.

Job Execution

Job and data needs be wrapped into a single package which is serialized to binary and put into a field of the request message. In our reference implementation, a package is compiled into DEX files and put into the JAR package. When receiving the package, a Worker deserializes and loads it using *DexClassLoader* from Android SDK.

4.4.4 Estimating Execution Time in Multiple Requests

Next, we build a simple estimation model to theoretically evaluate the performance of the system in both *synchronous* and *asynchronous* modes by comparing with the local execution measured on the starting device (Requester). To avoid complex cases, we assume dispatching a job with simple linear algorithm interacting solely with attached data without requesting any external hardware resources.

Single execution

Assume that a Requester installed on the device that has resource capacity RL_C (constant - measured before the execution), runs the job J having data processing speed γ (bytes/sec) locally on data with size M_J in total time T_J. The data processing speed γ is previously measured by running J on the device in a perfect condition (e.g. device is idle). By considering these parameters: γ, M_J and RL_C for each job on each specific device, we observe that T_J is proportional with M_J and inversely proportional with RL_C and γ, in other words:

$$T_J = \frac{1}{\gamma}(\frac{M_J}{RL_C}) \tag{4.3}$$

If we offload the job J to the P2P network with serve of N Workers ($N \geq 1$), the total execution time would be the sum of execution time of the slowest Worker, as they are working parallelly, plus the time data traveling between the devices in both directions and overhead time at the Broker. Regarding network transmission, our results in Figure 4.5 and [81] reveal that transmission time is linear with the amount of traveling data. Particularly, the transmission time would be $T_{Net} = \frac{M_J}{\alpha}$ where α is the speed of WFD (bps) at the moment. Since the job request and data must travel in the network from the Requester to Broker, Broker to Workers and be back, it would take totally $4T_{Net}$ for transmission time if we consider the same amount of the results. Finally, if we assign H to the overhead time at the Broker, the total remote execution time T'_J can be estimated as follows:

$$T'_J = \max_{i=\overline{1,N}} \{\frac{M_i}{\gamma_i RL_i}\} + H + 4\frac{M_J}{\alpha}$$

By combining with Equation 4.2 and set $S_{RL} = \sum_{i=\overline{1,N}} RL_i$ we have:

$$T'_J = \frac{M_J}{S_{RL}} \max_{i=\overline{1,N}} \{\frac{1}{\gamma_i}\} + H + 4\frac{M_J}{\alpha} \tag{4.4}$$

According to the strategy in Section 4.4.1, since we only select the Workers having better RLs than the Requester's, therefore $RL_i \geq RL_C, \forall i = \overline{1,N}$, so $S_{RL} = \sum_{i=\overline{1,N}} RL_i \geq N \times RL_C$. Also, in the perfect condition where all the Workers are fully dedicated for one job, we can consider $\gamma_i \geq \gamma, \forall i = \overline{1,N}$. By applying to Equation 4.4, we have:

$$T'_J \leq \frac{T_J}{N} + H + 4\frac{M_J}{\alpha} \tag{4.5}$$

From this equation, if we can offload a job with very small amount of data (e.g. sorting-list job, array data 1KB) on 3 available Workers ($N = 3$), we can consider the network transmission time and Broker's overhead is insignificant, then we obtain remote execution time $T'_J \simeq \frac{T_J}{3}$ which is 3 times faster than local execution.

Multiple executions

According to the Equation 4.3, if we start M jobs synchronously on M Requesters, the total time would be:

$$\sum_{i=\overline{1,M}} T_{J_i} = \sum_{i=\overline{1,M}} \frac{1}{\gamma_i}(\frac{M_{J_i}}{RL_{C_i}}) \tag{4.6}$$

If we offload all these jobs to the Broker and assume that they are resolved immediately by the Broker, they will be executed simultaneously and the total time to run all of them would be:

$$\sum_{i=\overline{1,M}} T'_{J_i} = \max_{i=\overline{1,M}} \{\frac{M_{J_i}}{\gamma_i S_{RL_i}}\} + \sum_{i=\overline{1,M}} (H_i + 4\frac{M_{J_i}}{\alpha}) \tag{4.7}$$

Where H_i is the overhead the Broker must pay to pick and forward the request $\#i$ to the Workers, N_i is the number of Workers participated in resolving the request $\#i$. By applying the same analysis as in Equation 4.5, we have:

$$\sum_{i=\overline{1,M}} T'_{J_i} \leq \max_{i=\overline{1,M}} \{\frac{T_{J_i}}{N_i}\} + \sum_{i=\overline{1,M}} (H_i + 4\frac{M_{J_i}}{\alpha}) \tag{4.8}$$

4.4.5 API Usage Scenario

Distributed application programming often terrifies the average developer with heterogeneity of components and technologies, especially in mobile platforms. Therefore, simplification of programming model for easy integration to applications is always a mandatory requirement.

```
new Requester(this,IP){
  @override
  public void messageReceived(int type,byte[] msg){
    switch(type){
      case TASK_INFO:
        writeLog((String)msg);
      case TASK_DONE:
        displayBitmap((Bitmap)msg);
    }}
  @override
  void send(){
    Job jobDef = new JobImpl());
    byte[] data = loadDataFromFile(dataPath);
    JobMessage msg = new JobMessage(jobDef,data);
    msg.addDataParser(new DataParserImpl());
    sendJob(msg);
  }}.execute();
```

Code Snippet 4.1: Overriding and initiating Requester

Upon adopting our middleware library, developer can easily integrate these components into their applications and initiate the system. As we discussed before in Section 4.4.1, our middleware enables multiple network topology by using different combinations of the the components, for example, the simplest form can be illustrated as in the Figure 4.1. Assume we are building a collaborative network including 6 mobile devices connected in one WFD network. First of all, we integrate all Broker, Worker and Requester components into the application because the role of each device will probably change if network changes, the device can firstly play role of Requester and later Broker.

Since Broker and Worker are the prebuilt components, which only require the simple initiations, developer only needs to override the Requester class to define what to offload and how to handle result (Listing 4.1). Particularly, the `messageReceived()` method must be overridden to listen to the messages including system information under `TASK_INFO` label and task results under `TASK_DONE` label, where `msg` contains the detailed messages or result data returned from Broker. Finally, developer overrides the `send()` method to define `job`, `data` and `dataParser` respectively and calls `sendJob` to pack all the content into the JAR package and dispatch to the Broker.

4.4.6 Failure Handling

In mobile opportunistic networks, connection failures between devices may happen at any time due to the nature of volatile wireless network communications [82]. Therefore, it is necessary to ensure that a job is executed and its result is successfully returned back to a client. We distinguish two failure cases: (1) a failure happens on a Worker and (2) a failure on a Broker. Then, we designed the *Signal Server* module running on a Broker and *Signal Client* running on the both Requester and Worker to exchange status signals. Once a Requester and Workers are connected to a Broker, they open a signal communication channel to exchange their status. By default the update status is exchanged every 10 seconds[5].

The *Signal Server* module on the Broker stores the list of Workers for further reference. If a device doesn't receive any signal during the timeout period, its peer could be lost. If

[5]this property can be updated at the system initiation

the Worker is disconnected, it will quickly terminate all current executions and connections and turn into the *waiting state*. Meanwhile, the Broker detects the Worker's failure, finds the tasks with the similar *Worker ID*, and runs the failed job locally. Furthermore, the Broker also removes the Worker having the similar *Worker ID* from the Worker list. In the second case, if a failure occurs on the Broker, it removes all the related jobs and connections. Meanwhile, all the peers including Requesters and Workers terminate corresponding connections with the Broker and turn themselves into the *waiting state*. In this case, the Requesters simply executes the failed jobs locally.

4.5 Evaluation

We have evaluated the effectiveness of our approach through micro-benchmarks and a realistic case study. The testbed for experiments has been built up with various Android devices featuring WFD and one server for comparing with performance of traditional approaches. Table 4.1 shows the mobile devices and a PC used in the experiments.

Table 4.1: List of devices used in the experiments in chapter 4.

Device	**CPU**	**RAM**	**Battery**	**OS**
Moto G4	Octa 1.5GHz	2GB	3000mAh	6.0.1
G4	Quad 1.5GHz	3GB	3000mAh	5.1
Asus ZF2	Quad 1.7GHz	3GB	2400mAh	5.1
BLU R1	Quad 1.3GHz	1GB	2500mAh	6.0
S3	Quad 1.4GHz	1GB	2100mAh	4.4
Dell PC	i7 3.6GHz	8GB	N/A	Win10

4.5.1 Micro Benchmarks

We measured the system overheads such as network initialization time and message transmission/processing time through a set of micro-benchmarks.

Discovery Phase

First, a WFD-enabled mobile device needs to search nearby devices by scanning available devices. Since the peer discovery phase is mandatory for communications [29], we measured the impact of the discovery process on each device. Figure 4.4-Top shows the average time that each device discovers nearby devices. According to the results, it takes a few dozens of millisecond, particularly: 10ms on G4, 19ms on Asus ZF2 and 14ms on Galaxy S3, which are considerably short compared to the overall execution time.

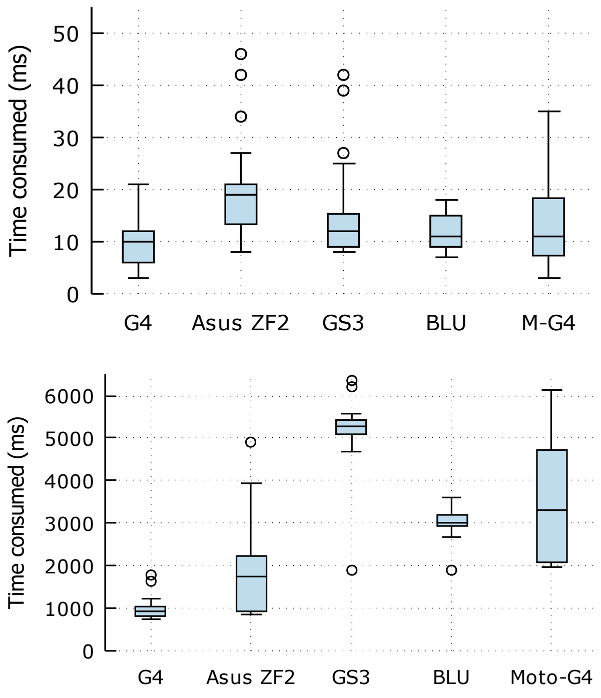


Fig. 4.4: Average discovery and connection establishment time.

Connection Establishment

After the discovery, the two devices start a negotiation to ordain one to become a group owner (GO) [5]. The GO will take the role of a virtual AP that initiates DHCP to allocate a client IP to the connecting devices and forms up a group.

Due to the complexity of establishing connection, mobile devices have to spend much more efforts for this process than the discovery. Figure 4.4-Bottom shows the variability of connection initiation time on three devices when connecting to the same GO; the average time on G4 is 1024ms, on Asus ZF2 is 1822ms and 5680ms on Galaxy S3, respectively.

Then, we experimented with WFD-related processes including the peer discovery and connection establishment phase. Since the peer discovery and connection establishment phases are mandatory for a peer-to-peer network, we measured the impact of these processes on each device. Figure 4.4 shows our experimental results. In particular, the discovery phase normally took 10—20ms on most devices, while WFD connections were established within 1—2seconds on high-end devices (e.g., G4, Asus ZF2, etc.) and 3—5 seconds on low-end device (e.g., Galaxy S3, Blue).

Message Dispatching Time

Next, we measured the message delivery time that greatly affects total execution time and energy consumption in accordance with different network conditions such as delay/bandwidth characteristics and signal strength. For the experiment, we pre-installed the middleware system on three devices. One device hosted both the Requester and Broker, and the others operated as Workers. For accurate measurements, we firstly synchronized the clocks of all the devices using a NTP server, then recorded timestamps on each device. As a benchmark application, we implemented a benchmark program where the Requester simply sends out a message (i.e., job definition) to these Workers through the Broker and then each Worker returns an acknowledgment back to the Requester. Figure 4.5 shows the incremental distribution time among the devices when sending packages with sizes increasing from 1KB to 1.5MB.

Middleware Overheads

To estimate the overheads of our middleware implementation on top of ZeroMQ library, we compare its performance with ZeroMQ's performance by measuring the round-trip time of various size data packages traveling from the first device to the second and third ones and back. The total time is calculated as the sum of processing time on each device plus the transmission time. According to the Figure 4.6, the overheads of our implementation fluctuate from 21 to 33% more when sending packages with sizes increased from 1KB to 1MB.

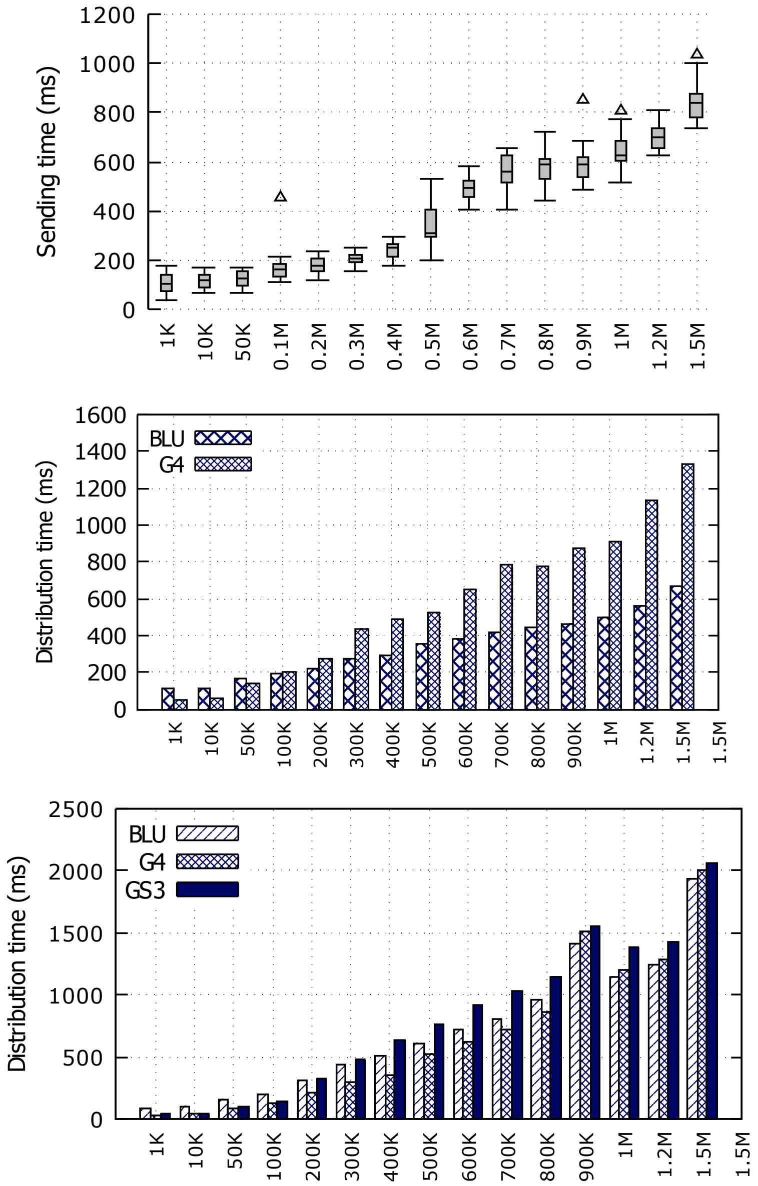


Fig. 4.5: Time when sending various jobs to multiple devices.

4.5.2 Case Studies

In this section, we evaluate our approach through two case studies: Image processing and Failure Handling.

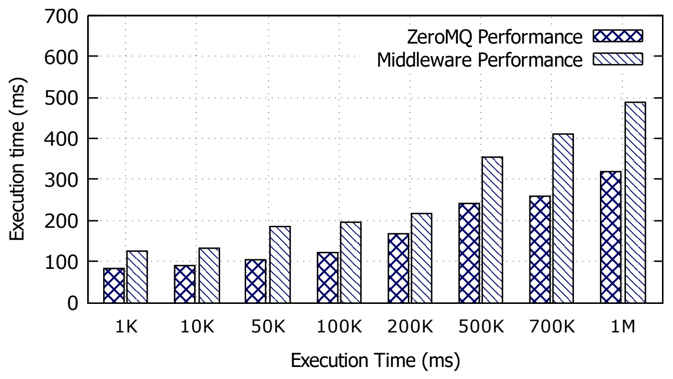


Fig. 4.6: Performance comparison of ZeroMQ and our middleware.

Video Sharing

The previous benchmarks show that our middleware can send 100KB message in approximately 100ms, thus technically we can stream video at acceptable frequency of 10fps with each frame contains 100KB of data.

In this application, we will stream an MP4 video file from one device to the others. FFmpeg Media Metadata Retriever library (FMMR) [6] is embedded to extract frames out of the video every 100ms to reach the speed of 10fps (Figure 4.7).

Theoretically, by integrating our middleware with above FFmpeg-supported library, it is possible to stream video data from any sources: Youtube, Vimeo or capture devices.

Image Processing

In this experiment, we implemented a simple image blurring application whose functionality implemented using Gaussian convolution was executed remotely. For the experiment, we first started a Requester sequentially on each of the two devices—low-end device (BLU) and high-end device (LG-G4). For each case we deployed consecutively one to four Workers to see how system compares and selects the appropriate Workers. According to the results that depicted in the Figure 4.8, if the job is offloaded from the BLU, all 4 Workers are selected and they help improve the performance up to 57% compare with the local execution.

[6]FMMR: `https://github.com/wseemann/FFmpegMediaMetadataRetriever`

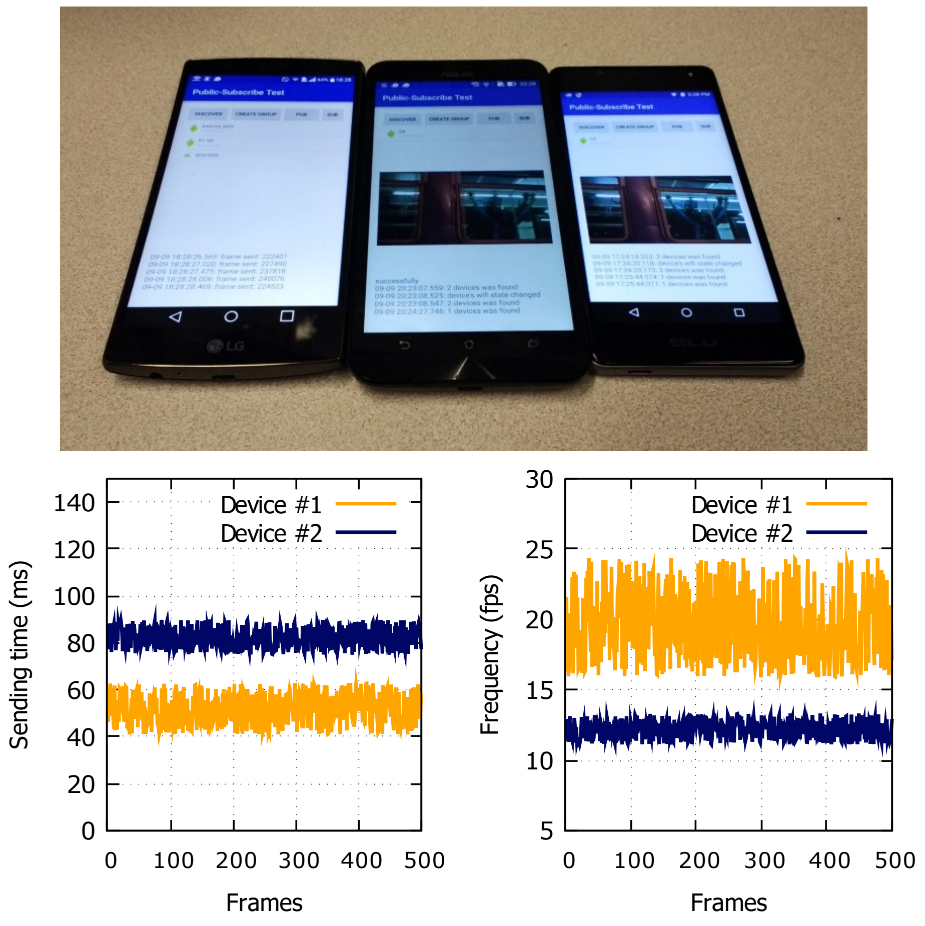


Fig. 4.7: Sharing video across mobile devices

Meanwhile, if the job is offloaded from the LG-G4, only the Moto-G4 and Asus ZF2 which have better *RL* values are selected, and they help improve the performance up to 35%.

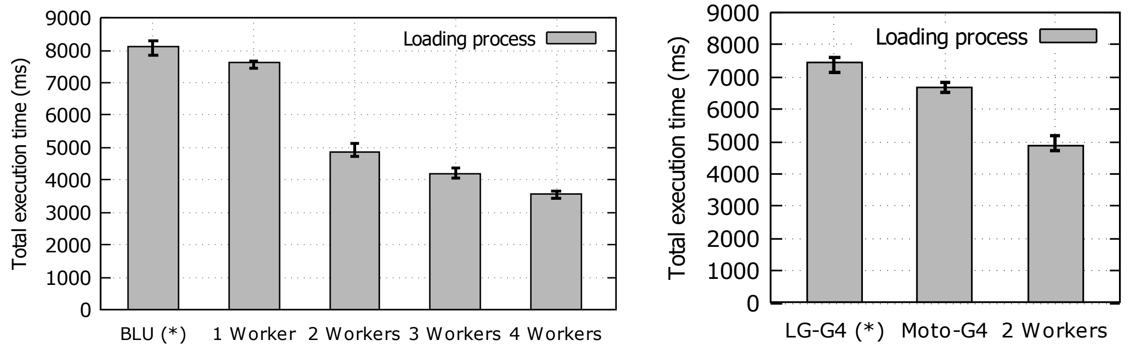


Fig. 4.8: Image processing results initiating from the different devices.

Multiple Requests In the previous experiment, we observed the performance benefit of the system in the *synchronous mode* where each job is only dispatched after the previous one has been completed. In this experiment, we will test the system in *asynchronous mode* by starting two Requesters at the same time to simultaneously send their jobs to the Broker. Then, we compare the cumulative execution time of running two jobs in the two modes. Figure 4.9-Top reveals that running two jobs in *asynchronous mode* help reduce the total execution time from 40 to 48% in the networks with one to four Workers. Compared with the result from Figure 4.8, it shows the average speed of running two simultaneous jobs is very close to single job.

Unlike the result obtained from running two jobs, running three jobs in *asynchronous mode* helps reduce execution time from 58-62% as shown in Figure 4.9-Bottom. The execution speed of running three jobs is also close to the speed of running one single job and two asynchronous jobs; the leftover, as we noticed, comes from the system overhead when it handles several jobs at a time.

Failure Handling

To evaluate our failure handling mechanism, we implemented a simple test application with one Requester, one Broker and two Workers for two sub tests. In the first test, we disconnect one Worker so that the assigned task can be executed on the Broker instead. Then we measured the final execution time of the complete job at the Requester. In the second test, we disconnected the Requester from the network and observed how our approach handles the exception and recovers the execution locally.

Figure 4.10 depicts the difference between the three processes: (1) normal process when all components collaborate successfully (yellow line), (2) failure handling process when failure occurs on Broker (blue) and (3) when failure occurs on Worker (green). The blue line reveals that the handling process takes longer execution time than normal process since the Requester needs time to detect and cover failure from the network. However, in this case the detection time only takes 1.7 seconds (timeout is set at 2 seconds) and the failure covered by a local execution is immediately started within 2-300ms. When the failure

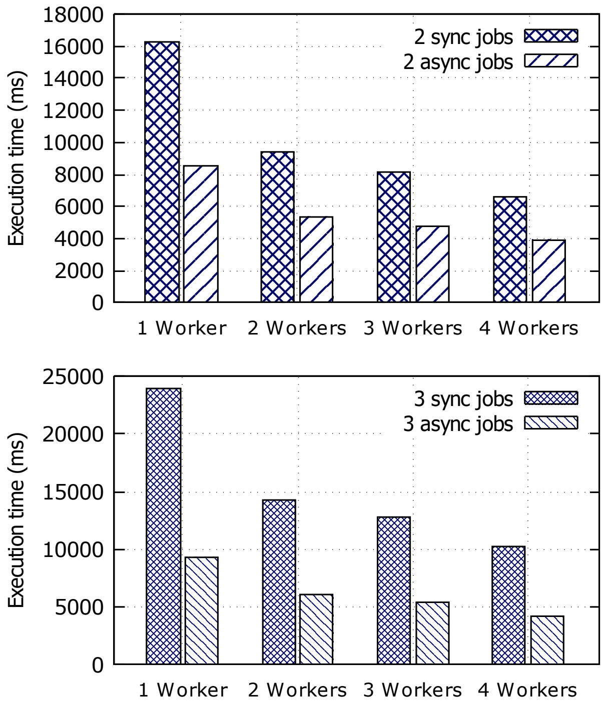


Fig. 4.9: Compare the cumulative execution time of the system between synchronous and asynchronous modes.

occurs on the Worker, it would take more time for the job to be done than the Broker's failure because the Broker must handle both failure detection and result transmission.

4.6 Related Work

Message-oriented middleware architecture such as ActiveMQ or DataTurbine [83] are only suitable for the development of distributed systems on stationary servers, their architectures and libraries are overwhelmed for the mobile network systems. Moreover, to the best of our knowledge, none of the up-to-date middleware these days support for both multiple remote execution delivery and device selection optimization on mobile networks.

There are a few similar attempt to address the compatibility problem. Fram [84], a publish-subscribe middleware for opportunistic content distribution on both wireless ad-hoc domain and wired Internet. It enables JMS-like simpler API and abstracts the heterogeneity of networks including BLE, WiFi Aware and LTE-Direct from the applications. ICD2D [85], implemented on NS3, is an information-centric device-to-device network which is distributed

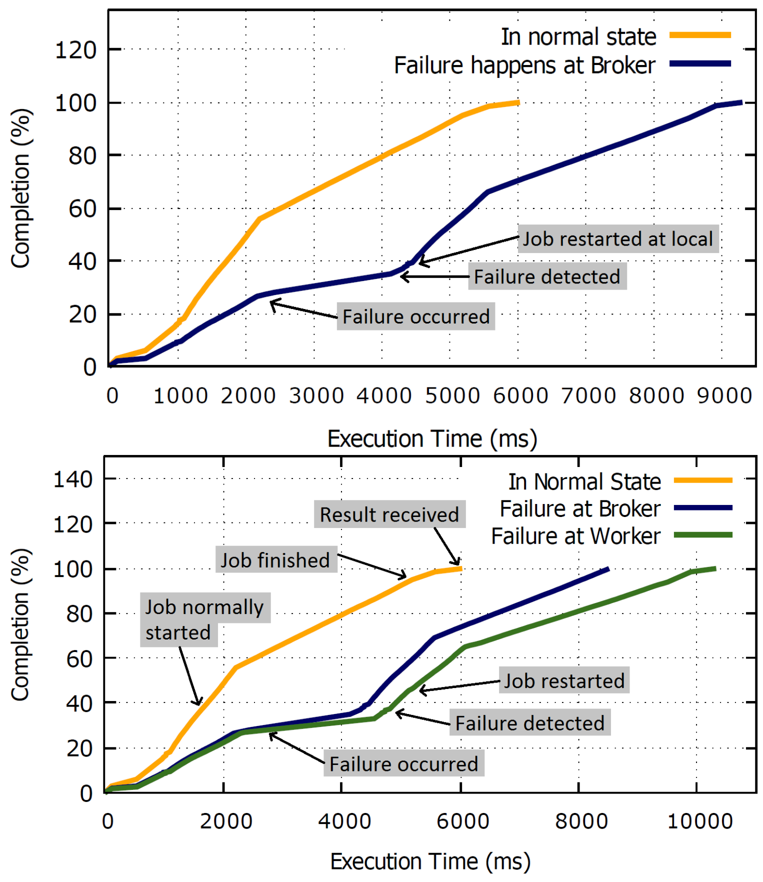


Fig. 4.10: Compare the normal process when all components work smoothly and when failure happens in image processing test.

and requiring no centralized coordination.

Our research shares the interest in the domain of remote execution on mobile ad-hoc network for performance enhancement. Serendipity [1] introduces a different job distribution algorithm based on PNP-blocks which each contains pre-process program to split input data into multiple equal segments and allocate to the parallel tasks. However, the limit of Serendipity's approach is that tasks are equally offloaded to the nearby peers without concerning about their capability, as well as lack of a novel mechanism for picking up some appropriate devices for a specific remote execution because all devices are not always available at that moment. As the extension of Serendipity, COSMOS [86] provides computation offloading as a service to mobile devices by efficiently managing cloud resources, allocating and scheduling offload requests as well as risk control.

Publish-subscribe architecture has a long history in distributed computing research with multiple paradigms [78]. Many of them focused on different angles of the problem: social network, network intermittent [87], mobility [88]. There are a few remarkable researches on mobile networks: SocialCast [87] is a routing framework exploiting predictions based on metrics of social interaction, which addressed the intermittently-connected human networks to improve performance and overcome high rates of mobility and long-lasting disconnections.

By leveraging device-to-device communication on publish-subscribe pattern, our software can serve for code offloading over multiple nearby devices without the need of Internet connection. According to this, our efforts have made an improvement of previous works like MAUI or COMET [3, 4] which ordinarily address offloading to the cloud servers.

4.7 Conclusion

In this chapter, we introduced a new middleware system which employs multiple messaging paradigms for computation offloading in opportunistic networks. In particular, we utilized the publish-subscribe paradigm to support multiple clients and remote executions concurrently. As a result, our system can flexibly allow any mobile device to offload code to nearby mobile devices by adapting mutable topology. The experimental results implemented in Android, show that our approach can effectively build a flexible and reliable peer-to-peer network using the WiFi Direct technology.

CHAPTER 5
JSREX: AN EFFICIENT JAVASCRIPT-BASED MIDDLEWARE FOR MULTI-PLATFORM MOBILE PEER-TO-PEER NETWORKS

[8] [1]

5.1 Abstract

Code offloading on mobile platforms has received much attention as a way of relieving heavy workload by utilizing power of the other devices or cloud servers. The offloading mechanisms rely on multiple platforms to be enabled on variety of mobile devices. To support platform heterogeneity, the execution code should be implemented in JavaScript and executed on the designate devices by correspondingly compatible JavaScript engine. In this chapter, we present a novel distribution mechanism with a built-in JavaScript execution package, annotation processor and an engine to enable code offloading among the devices and servers in peer-to-peer (P2P) networks. This approach includes a set of constraints for code implementation so developers can easily integrate to their project. Our evaluation, based on a testbed with Android and Windows Phone devices, demonstrates the efficiency of offloading JavaScript-based packages on multiple devices, as well as compares the performance between JavaScript and native versions.

5.2 Introduction

Along with rapid advancements in mobile hardware, mobile applications are becoming more sophisticated and computationally intensive. Nevertheless, certain applications, like those that attempt to do real-time speech recognition, image processing, language translation, or video rendering are facing a computational-resource scarcity problem. Furthermore,

[1] Minh Le, Stephen W. Clyde @ iiWAS 2017

we believe that the growth of computationally intensive mobile apps will continue to outpace advances in mobile computing power. This means that the resource scarcity will likely be an on-going challenge for the foreseeable future.

Li et al. address resource scarcity by offloading CPU-intensive computations to powerful cloud servers [9]. However, this approach requires persistent network connections, can incur cloud-related expenses [37, 89, 90], and is not suitable for wide-range mobility. Another approach is code migration within a mobile P2P network [91, 92], which allows one device to offload certain software components (e.g., functions or classes) to remote devices and to coordinate their execution via message passaging over WiFi-Direct [2], Bluetooth, or NFC. Despite its flexibility, however, this approach struggles with platform heterogeneity, a reality in today's fast moving mobile-device market (See Section 5.6).

This chapter proposes a new approach for offloading computation that supports device heterogeneity for a P2P code migration (See Figure 5.1). It consists of a middleware layer, called *JavaScript Remote Executor* (*JSReX*) that accepts requests from applications, then breaks up them up into small self-contained jobs, and coordinates their executions among multiple peer devices. Each job represents a computational sub-task and includes the necessary code for its executing, a subset of the request's overall data, plus meta-data for parsing and interpreting the data (See Section 5.4).

To overcome the limit of the prior JavaScript-based approaches which are only applicable for mobile web applications [9, 24, 93–95], as well as support native applications, we propose a new method that helps automatically convert native code (Java) portions into JavaScript by declaring method-scope annotations. If a method is attached with our annotation, its code will be translated into JavaScript during the compiling process.

In the next Section, we describe the technology that underlies our implementation. Section 5.4 discusses key issues in architectural design and implementation of (*JSReX*). Section 5.5 provides an initial evaluation of *JSReX* based on a micro-benchmark and three case studies that demonstrate its feasibility and efficiency on real devices with non-trivial

[2]http://www.wi-fi.org/discover-wi-fi/wi-fi-direct

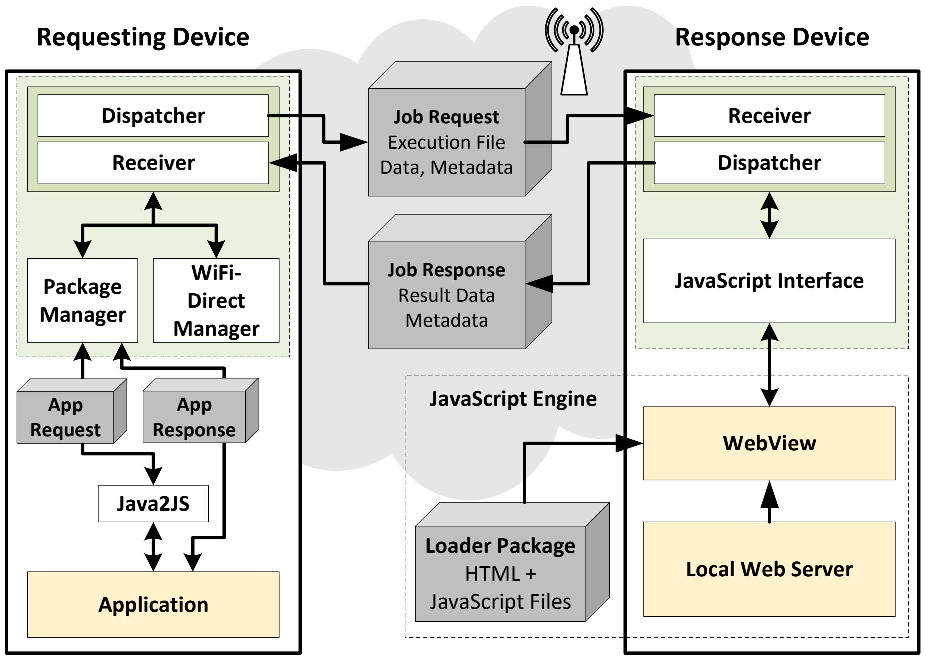


Fig. 5.1: An overview of our middleware

application requests. Section 5.6 then provides some additional insight on related work and Section 5.7 provides a conclusion with a statement about future work.

5.3 Background

In this section we describe three main technologies that we are going to use in this chapter, WiFi-Direct, Code Offloading and Local Web Server.

5.3.1 WiFi-Direct

Wi-Fi Direct [3] is a certification mark for devices supporting a technology that enables Wi-Fi devices to connect directly, making it simple and convenient to do things like print, share, sync and display. WiFi-Direct allows devices to communicate regardless of access point within longer distances (up to 200 meters) than the other technologies such as NFC [4] or Bluetooth as well as operate at much higher speed, 250mbps in comparison with 25mbps of Bluetooth 4.0 and 424kbps as of NFC. Moreover, since WiFi-Direct is built on top of

[3]WiFi-Direct: https://www.wi-fi.org/discover-wi-fi/wi-fi-direct
[4]NFC: https://www.androidpit.com/what-is-nfc

WiFi network interface, it is available on any WiFi-enabled device running Android 4.0 or higher without requiring hardware support.

Wi-Fi Direct communication always occurs within a single group where one peer in the group acts as Group Owner (GO) and the other devices become clients of GO. Such roles within the group are not predefined, but are negotiated upon group formation. After the GO is elected, the role of each peer remains unchanged during the whole group session. Only when the GO leaves the group, the peers become disconnected and a new group must be created [29]. Many studies take advance of WiFi-Direct to build low-level device-to-device communications [43, 44], while the other attempt to extend the distance limit of this technology by employing group-to-group mechanism [29].

5.3.2 Code Offloading

Offloading is an opportunistic process that relies on remote servers or platforms to execute code delegated by a mobile device. In this process, the mobile application has local decision logic to detect resource-intensive portions of code (e.g. time- or CPU-intensive), so given the current network condition the mobile application can estimate where the execution of code will require less computational effort (remote or local) to enhance the performance and save energy [91, 96]. Code offloading is generally categorized by the three questions, *what to offload*, *when to offload* and *where to offload*. Several research address *what to offload* by employing *code profiler* [97, 98] or *annotations* to define which portion of the code to be offloaded. For the question *when to offload*, the middleware determines the most appropriate time or whether to offload by comparing performance of the remote cloud server versus local device based on the status information of many factors including network, algorithm of the execution code or cloud's and local device's capabilities [90]. Finally, for *where to offload*, one should be able to choose the best platform or server for a specific task or know how to break the task's logic into sequent execution parts to offload [99].

(*JSReX*) addresses the question *what to offload* by adopting JavaScript-based execution package for offloading code to facilitate multiple platforms. Specifically, for each common platform, such as Android and Windows Phone, (*JSReX*) includes a JavaScript engine that operates in conjunction with a local web server.

5.3.3 Local Web Server

There are several ways to load JavaScript (JS) code into the memory but the most popular is using either (1) JS native compiler (e.g. Rhino [5]) or (2) default web browser. The benefits of the JS native compiler is the execution code will be loaded and parameters are injected directly without going through a middle interface. However, according to our evaluation, the speed of loading JS by this method on Android and Windows Phone is much slower than by using the default web browsers (Section 5.4.4).

In contrast, loading JS code on a default web browser requires an external web server to load a *trigger HTML file* which is stored on local storage, and this file in turn loads the JS execution code. A local web server is an embedded HTTP server library that can be integrated in applications to provide simple request/response services for the host. An example of local web server is NanoHttpd for Android applications [6]. By using a local web server, a developer can avoid deploying a separated external web server to load HTML and related resource files; place files to anywhere in the device's local storage minimal constraints; and avoid problems arising from firewalls or access controls.

5.4 Approach

(*JSReX*)'s design and implementation satisfy five important objectives: heterogeneity, robustness, simplicity, reliability, and efficiency. For heterogeneity, *JSReX* ensures that common devices, like Android or Windows phones, can join the network and execute jobs. For robustness, it supports a board range of computational requests from virtually any kind of mobile app and for any kind of data. Also, *JSReX* supports a variety of network topologies, including closed-range and dynamic P2P networks, where devices don't have to be connected to Internet and can join or leave at any time. For simplicity, *JSReX* makes it easy for app developers to integrate computational offloading into their mobile

[5]Rhino: https://github.com/mozilla/rhino
[6]NanoHttpd: https://en.wikipedia.org/wiki/NanoHTTPD

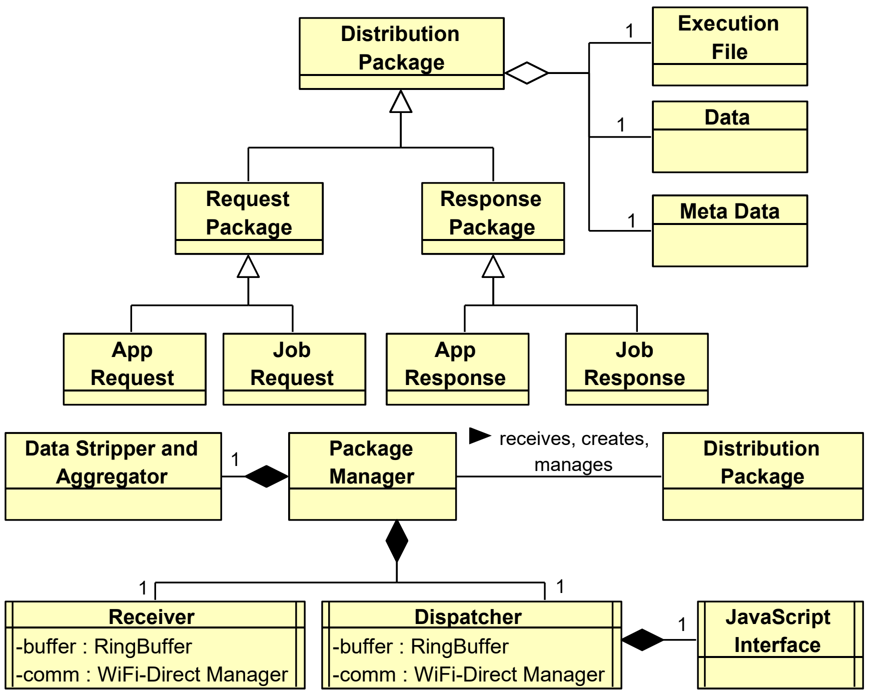


Fig. 5.2: The major components of our middleware.

apps by allowing them to code request computations into a simple JavaScript template. Finally, for reliability and efficiency, (*JSReX*) utilizes standard communication protocols and JavaScript engines.

5.4.1 Architectural Overview

Figure 5.2 shows the major components of *JSReX*. At the heart of the system are *Distribution Packages* that represent either requests for remote execution or responses to those requests, at either the application or job level. An *App Request* represents some chunk of work that a mobile app would like to be done by its peers and an *App Response* is the aggregated result from the peers. Similarly, a *Job Request* represents a sub-task for one peer to execute and a *Job Response* is the result.

Every *Distribution Package* includes three items: an *Execution File*, *Data*, and *Meta-data*. The *Execution File* is a JavaScript code snippet written by the application developer following the template shown in Listing 5.1. By having the necessary computation implemented in JavaScript, *JSReX* can enable a wide range of heterogeneous peers to volunteer to execute jobs. The data in a distribution package is either input or output data, depending on whether the package is a request or response. The *Metadata* describes the structure and semantics of the data, enabling peers to parse and interpret the data as they execute their jobs.

In general, a *Package Manager* handles both message sending and receiving processes. It invokes a *Data Stripper and Aggregator* to break the *App Request* messages into smaller blocks and calls the *Dispatcher* to deliver to the peers. In the destination device, the *Package Manager* maintains a *Receiver* thread to receive incoming messages. A detailed message workflow will be described in the next section.

5.4.2 Java to JavaScript Conversion

Developers working with native code (e.g. Java) may find it cumbersome to prepare JavaScript execution packages. Therefore the prior approaches only emphasized web applications where they could offload JavaScript code directly. To address this issue, we employed JSweet library [7] and proposed a method to convert Java code snippets into JavaScript using annotations.

In the Java code, the developer places the `@ToJS` annotation before method prototype. During compilation, the associated *annotation processor* will combine the middleware libraries (e.g. `JavaScript Interface`) to generate a JavaScript file containing converting code of the method with conventional structure. Then, the developer will use the generated JS file as an execution code input to create an `App Request`. This feature is very useful for those who are unfamiliar with JavaScript, it also avoids being distracted by using JS in Java native code.

5.4.3 Workflows

An application developer can integrate offloading of certain computations into a mobile app by either implementing the computation in JavaScript, following the template shown in Listing 1, or putting `@ToJS` on a method to get the JS file automatically. When using the

[7] JSweet: http://www.jsweet.org

```
function entry(){
  if (window.jsInterface){
    var in=window.jsInterface.getParam('p','base64');
    ...
    var out=resolveData(in);
    ...
    window.jsInterface.returnResult(out);
  }
}

/* main resolving function */
function resolveData(data) {
  ...
}

$(document).ready(function(){
  entry();
});
```

Code Snippet 5.1: A sample of JavaScript template.

template, the developer will implement the execution code in the `entry()` function, which is will be started after JQuery [8] detects that the DOM [9] is fully loaded.

At the beginning of the `entry()` function, the compiler checks if an instance of *JavaScript Interface* (Listing 5.2) has been initiated through the `window.jsInterface` object, this is mandatory because the *JavaScript Interface* handles communications between the native and JavaScript interfaces. If the `jsInterface` object is not available, the execution code should be canceled.

To retrieve converted data from the native code, developer may call the function `getParam()` by providing input parameters name and convert type. In the above example, the value `in` will be assigned a value of parameter "`p`", encoded in `base64`.

After the code execution is done, the JavaScript will return results back to the native interface through the function `returnResult()`. Finally, the last three lines represent the

[8]JQuery: https://jquery.com
[9]DOM: https://www.w3schools.com/js/js_htmldom.asp

```
public interface JavaScriptInterface {
  public String getParam(String name,String type);
  public byte[] readFile(String uri);
  public String normalizeHtml(String content);
  public byte[] getDataFromUrl(String url);
  public String getTextFromUrl(String url);
  public void loadLink(String url,int numOfParts, int index);
  public void returnResult(String result);
  ...
  public interface ResultListener {
    public void resultAvailable(String result);
  }
}
```

Code Snippet 5.2: Functions of JavaScript Interface.

default entry point of JQuery which will automatically load the `entry()` function when DOM is ready.

Sending a request

A mobile application can then have peer devices execute some piece of work by creating an *App Request* containing this JavaScript file, the necessary data, and *Metadata*, and then sending it to the *Package Manager*. The *Package Manager* will invoke the *Data Stripper and Aggregator* module to analyze the request and create one or more *Job requests*, according to the number of available peers, it will also create a `placeholder` for this request and store in the *Placeholder Map* with an `unique request ID`. Then, the *Package Manager* distributes those *Job requests* to peer through the *Dispatcher*, which in turn uses the *WiFi-Direct Manager* for the actual network communications (Figure 5.3).

On each peer, a *Receiver* listens for incoming *Job Requests* and store them into a *Ring Buffer* [10]. When it is able to process a request, the *JavaScript Interface* pops the oldest *Job Request* from the *Ring Buffer* and launches the *Execution File*, which in turn interacts with a *WebView* to run the job.

[10] https://en.wikipedia.org/wiki/Circular_buffer

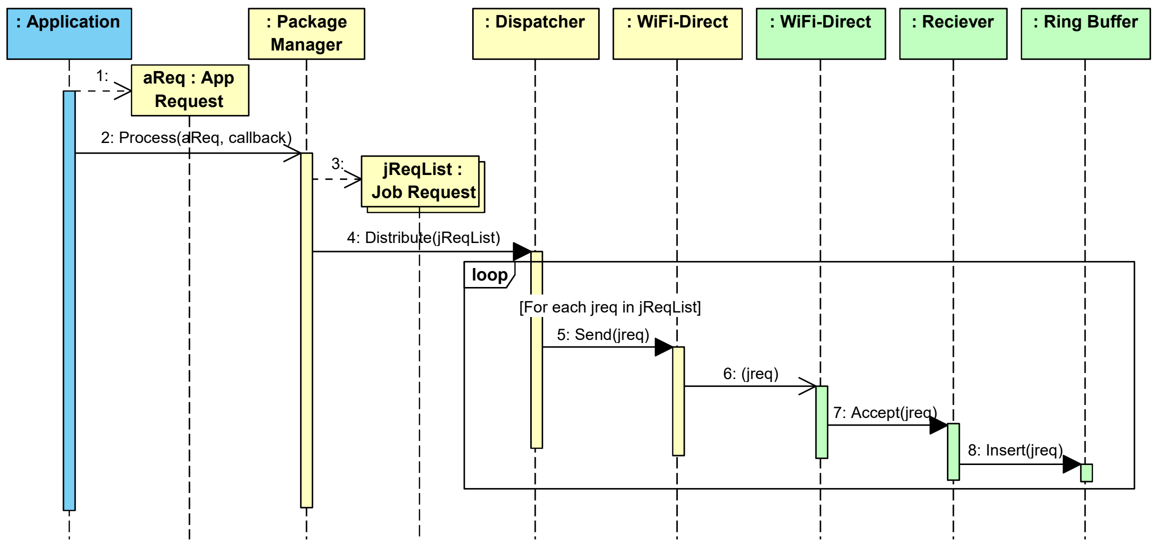


Fig. 5.3: The workflow that sends a request to the destination device.

To load the *Execution File*, the WebView opens the *Loader Package* and loads the trigger HTML file through the *Local Web Server*. While the HTML is loading, the necessary JavaScript libraries such as JQuery and RequireJS [11] are headforemost loaded, then come the execution file. The *JQuery* waits until the HTML load has been completed to invoke the `entry()` function of the *Execution File*.

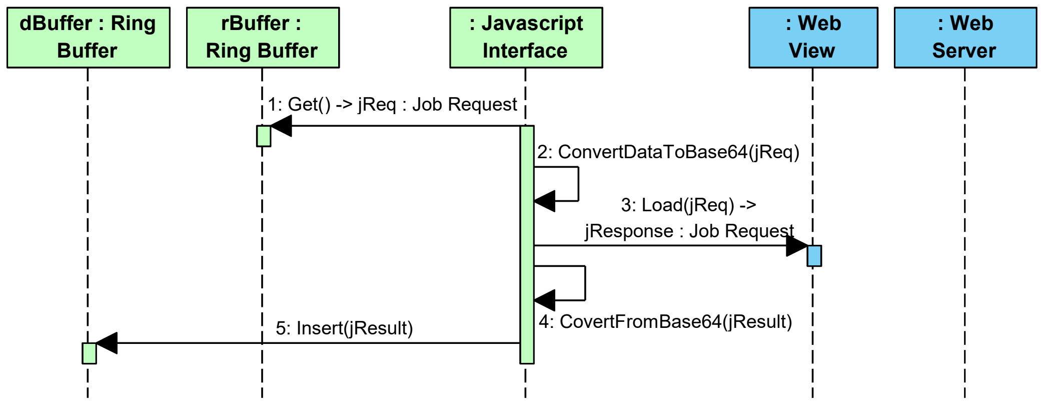

dBuffer : Ring Buffer
rBuffer : Ring Buffer
: Javascript Interface
: Web View
: Web Server
1: Get() -> jReq : Job Request
2: ConvertDataToBase64(jReq)
3: Load(jReq) -> jResponse : Job Request
4: CovertFromBase64(jResult)
5: Insert(jResult)

Fig. 5.4: The workflow that executes the remote task and return a result.

The *JavaScript Interface* converts job's data into `base64` string format and when it

[11]RequireJS: http://requirejs.org

receives result back, it converts them back from base64 to the original format, i.e., a binary format. Then, the result is packed in a *Job Response* and stored in the *Return Ring Buffer* (Figure 5.4).

Returning a response

Figure 5.5 describes the response message flow from a remote peer back to the request device. The *Dispatcher* on the peer periodically checks for responses queued in the *buffer*. When a response is available, it will be popped out and sent back to the requesting device via *WiFi-Direct Manager*. On the requesting device, the *Receiver* receives a *Job response* and saves into its *Return Ring Buffer*. The *Package Manager* will pop the response, find its ID and invoke *Data Stripper and Aggregator* to merge it to the corresponding `placeholder`. When the placeholder is fully filled with all partial responses, its contents will be merged into a *App Response* and this final response will be sent to the calling application through a callback function.

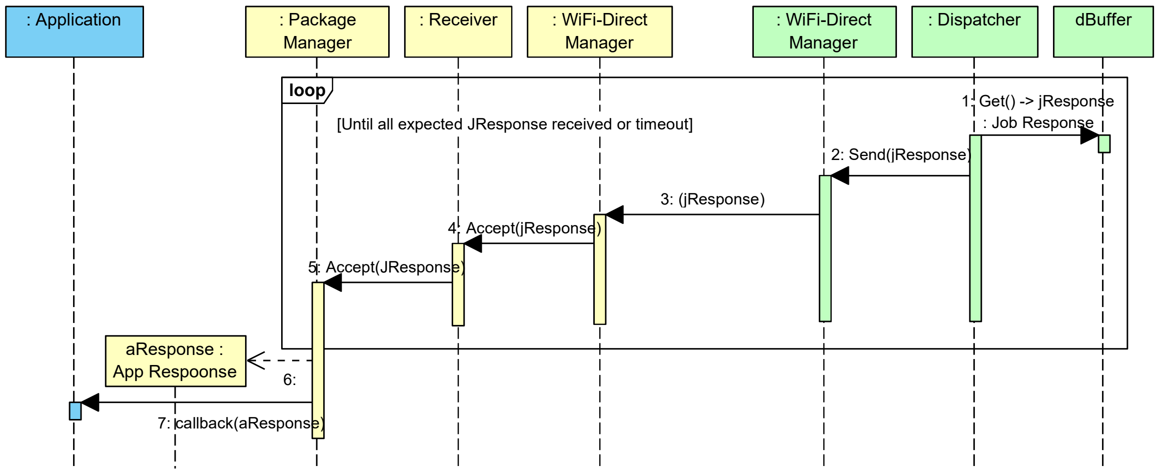


Fig. 5.5: The workflow that returns a result to the requesting device.

5.4.4 Run-time Considerations

Next, we discuss the (1) P2P communication, (2) template and metadata, and (3) JavaScript execution. We also provide an overview of concrete implementation of *JSReX* on the Android and Windows Phone platforms.

P2P Communication

To enable P2P connections, *JSReX* includes a *WiFi-Direct Manager* module to wrap up the default WiFi P2P feature. According to the WiFi-Direct standard [100], when connection is established, one device will be assigned as the Group Owner (GO) and the others will become clients (peers) of the GO's group. Then, the *WiFi-Direct Manager* on the GO will send status requests to the peers to collect information of them. One peer is considered as unavailable if it has battery level less than 20% or CPU usage at 100%.

Message Format

A general message contains a `Request ID`. To distribute an *App request* message to the peers, *Package Manager* breaks the message into a number of *Job requests* of which each is a copy of the message with partial data. It also creates a `placeholder` in the *Placeholder Map* with a key that is the `Request ID` (Figure 5.6). When a *Job response* arrives at the requesting device, *Package Manager* will use its `Request ID` to open the corresponding `placeholder` and fill it in to the appropriate position using its index in the `metadata`.

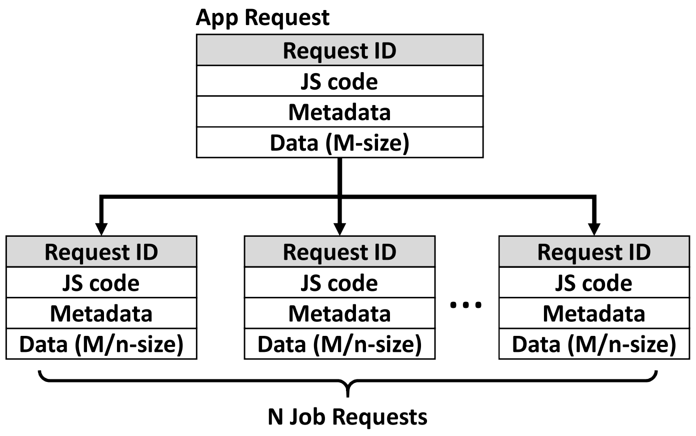


Fig. 5.6: The workflow that executes the remote task and return a result.

Metadata

To make data understandable for the automatic parsers on the response devices, distribution packages contain metadata that describe the structure and semantics of request data. The metadata is captured in a JSON file, referenced by either a file path in the package or an URL.

JavaScript Executor

To find an efficient ES5 compatible JavaScript engine on Android platforms, we embed and compared several different approaches: Rhino [12] (1), NanoHttpD [13] (2) and AndroidAsync [14] (3). Similarly, for Windows Phone platforms, we compared uHttpSharp [15] (4) and SimpleHttpServer [16] (5). All of the approaches, except Rhino, are essentially local web servers with an integrated WebView.

Figure 5.7 summaries the performance of all 5 engines when offload the word counter service (Section 5.5.2) on to a remote device. The Rhino JavaScript engine had the highest execution time (lowest performance), but can be used directly without a WebView [101]. Based on these results, we selected `NanoHttpD` for JavaScript engine on Android platform and `uHttpSharp` on Windows Phone.

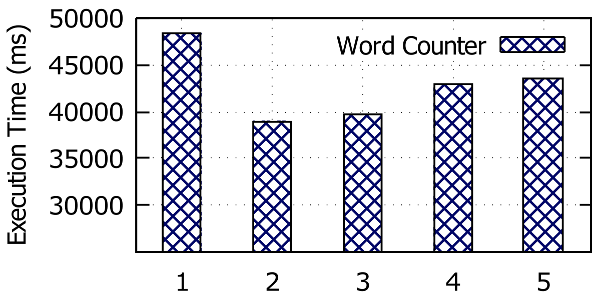


Fig. 5.7: Performance comparison of multiple JavaScript engines on Android and Windows Phone platforms.

[12]https://github.com/mozilla/rhino
[13]https://github.com/nanohttpd
[14]https://github.com/koush/androidasync
[15]https://github.com/bonesoul/uhttpsharp
[16]https://github.com/jeske/simplehttpserver

Figure 5.8 illustrates how JavaScript code is loaded by the default web browser. Firstly, *JavaScript Interface* pops out an *App Request*, extracts and saves the JS file to the same location with the *Loader Package* on the local storage. After that, it invokes the `load()` function of the web browser to load the HTML file from the *Loader Package*. This makes the browser send a HTTP GET request (in the format `http: //host/load.html`) to the *Local Web Server*. The web server will resolve the URL in the request and search for the HTML file in the *Loader Package*. This file includes a link of the JavaScript code, when it is loaded the JS code will also be loaded into the memory. Finally, when the browser completely executes the JavaScript code, it calls the `returnResult()` function to return results back to the middleware.

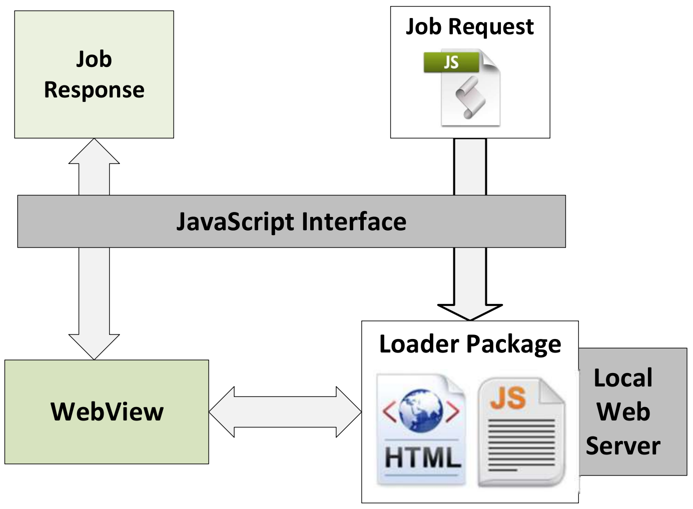


Fig. 5.8: How JavaScript code is loaded by the default web browser and a local web server.

To support multi-threading, we maintain a number of browser instances (the default number is 5 but it is customizable by a configuration file), each instance can resolve a request concurrently.

Runtime Failure Handling

Due to the highly intermittent characteristic of P2P mobile networks, failures can occur at any portion of the system. In *JSReX*, the *WiFi-Direct Manager* handles the system failures by exchanging status information between the peers over the detected status channels. The request device saves the copies of all the *Job Requests* in the local *Local Buffer* before dispatching. If a status channel fails for a peer, the *Package Manager* will find the corresponding *Job Request* from the *Local Buffer* and execute it locally. Moreover, *JSReX* uses a simple checksum mechanism to check data integrity. If package lost happens on any peer, the *Package Manager* will execute similarly the corresponding *Job Request* locally.

5.5 Evaluation

We built a test-bed with real devices (Table 5.1) to evaluate the effectiveness of *JSReX* through a micro-benchmark and case studies of three representative mobile apps and a P2P network of heterogeneous mobile devices.

Table 5.1: List of devices used in the experiments in chapter 5.

Device	**CPU**	**RAM**	**Battery**
LG Volt	Quad-core 1.2GHz	1GB	3000mAh
LG Tribute	Quad-core 1.2GHz	1GB	2100mAh
Moto G4	Octa-core 1.5GHz	2GB	3000mAh
BLU R1	Quad-core 1.3GHz	2GB	2500mAh
Lumia 550	Quad-core 1.1GHz	1GB	2100mAh
Dell PC	Intel i7-4790 3.6GHz	8GB	Wall-plugged

5.5.1 Micro-benchmark

To measure the overheads of the system, we offloaded an simple package which only has remote devices to sleeps for 2 seconds, so computational costs were predictable [102]. Figure 5.9 shows the performance of the system in terms of distribution time when sending a number of packages of varied sizes over the network. To isolate *JSReX*'s overheads, we performed this micro-benchmark on a P2P network with just two devices and removed the 2-second latency for the simulated computation from the timing measurements. The results showed that *JSReX* takes about 80ms on average for the background processing of each request. The rest of time is spent for data transmission over network and data conversion between the native and JavaScript code.

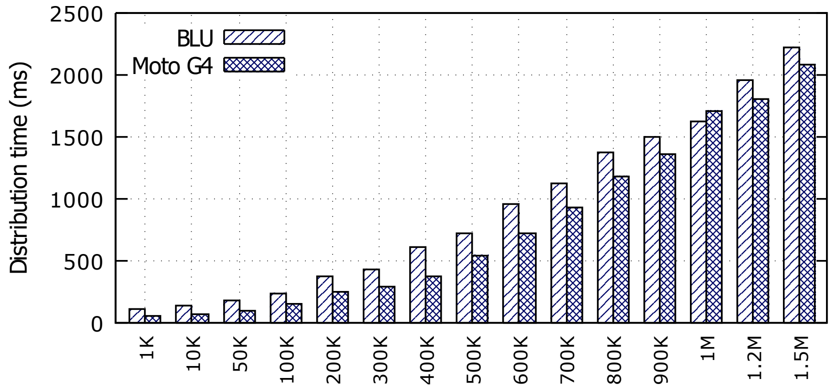


Fig. 5.9: System overheads on two mobile devices when sending packages of varied sizes.

We also excluded the discovery and connection establishment phases since they are very much dependable on device specs (Figure 5.10) and only involved in the system initiation and recovery. Thereafter, in our experiments we assumed the network was connected and only measured the total time starting from packaging (T_{pack}) on the requesting device, sending/receiving over network (T_{send}, T_{rec}) and resolving (T_{res}) on the remote peers.

$$T_{Total} = T_{pack} + T_{send} + T_{res} + T_{rec} \tag{5.1}$$

5.5.2 Case Studies

We integrated *JSReX* into several different mobile apps: image processor, word counter, pdf-to-image and cloud offloading. For each app, we measured the total execution on the requesting device, including all overheads, for multiple test cases and on P2P networks, with differing numbers of devices. Then, we compared with the native versions of the same apps to figure out where is the need of JavaScript. In the native version, we replaced the JavaScript *Execution File* by an Android DEX format file [17] with the same workflow and implementation. The DEX file could be loaded to memory by DEX Class Loader from Android SDK.

[17]DEX Format: https://source.android.com/devices/tech/dalvik/ dex-format

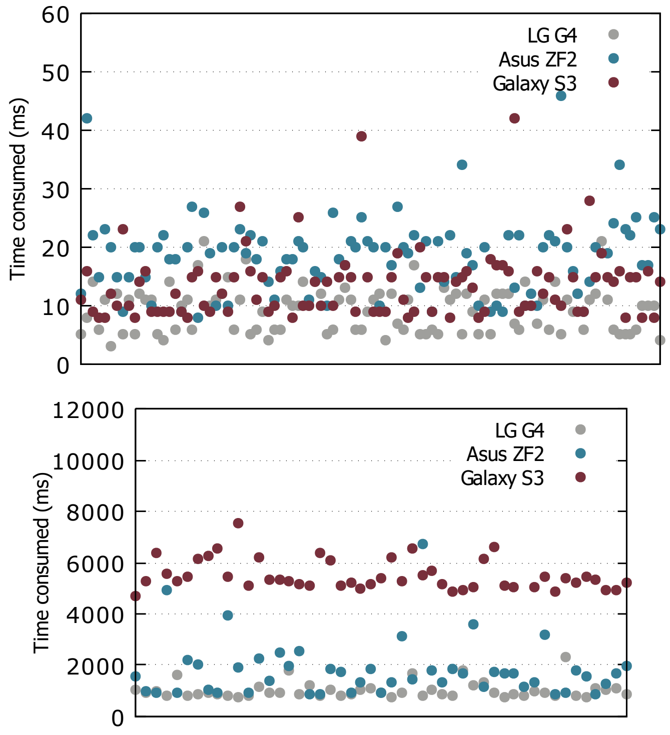


Fig. 5.10: Discovery time and connection time on different devices

Image Processor

This example app uses an image processing algorithm based on Gaussian Convolution [18] method to blur an image. The image is split into smaller parts by vertical cuts to distribute to the peers. Each of the peers executes the code on the partial image and returns the result to the requesting device (Figure 5.11).

We implement the Image Blur process in JavaScript and Java versions, Figure 5.12 reveals that the performance of both versions increase when adding more devices to the network. Particularly, with a big image sample (with size 3900 x 3200) from 12% (2 device) to 43% (4 devices) with JavaScript version and from 35 – 66% accordingly with Java version. It also shows the native version always prevails the JavaScript, from 16% in the case of single device up to 50% with 4 devices in the same configurations, however, the collaborations using JS by 3 more devices perform better than the native version in a single device for at

[18] Gaussian Blur: https://en.wikipedia.org/wiki/Gaussian_blur

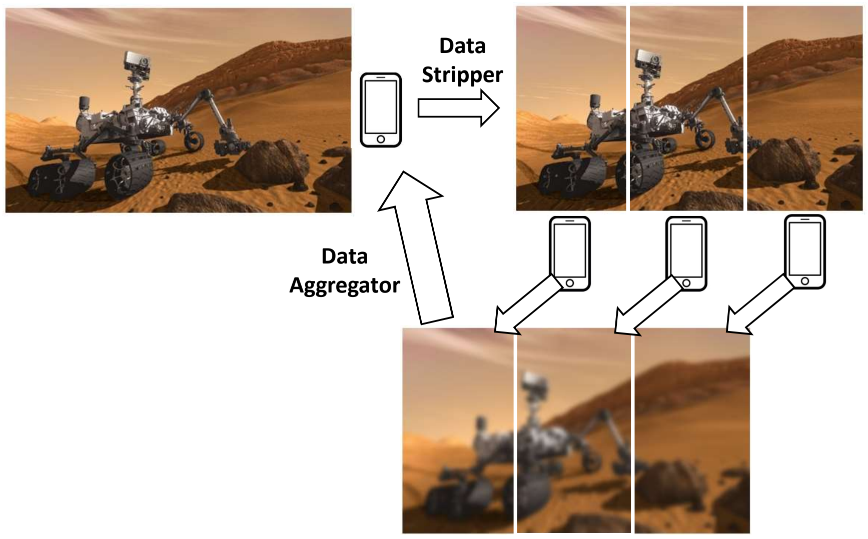


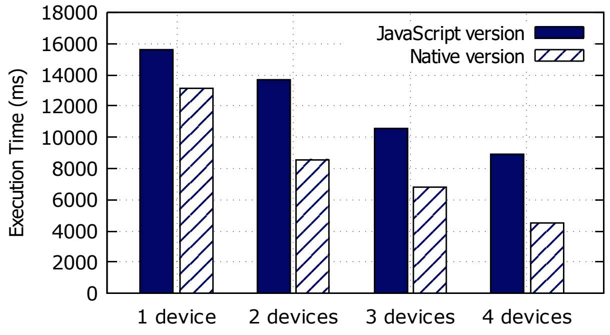
Fig. 5.11: Image processing overview.

least 20%. With a smaller sample (2500 x 1500), we achieved the increment from 31 – 54% and the performance of 2 devices is 23% higher than the native version on a single one (all numbers are rounded).

The reason for that overhead in JavaScript is that we have to convert the image data back and forth between binary and `base64` formats and perform Gaussian convolutions on base64 data which may take longer than on binary data. This is evident in the results, because the performance difference between JS and native versions is really distinct on the big sample (up to 37%), it's only 16% on the small sample. Also, the process of loading preliminary resources including the trigger HTML and JavaScript files somewhat delayed the execution, however, according to our experiments this process only takes 30-50ms for every execution, thus can be omitted.

Word Counter

The word counter uses a bruteforce algorithm to count the n most frequently appearing words in an online document, where n is defined by the request app and captured in the *App Request*'s metadata. The algorithm examines every word except the stop words (articles, conjunctions, etc.) and stores the number of word visits in a list. Finally, the algorithm



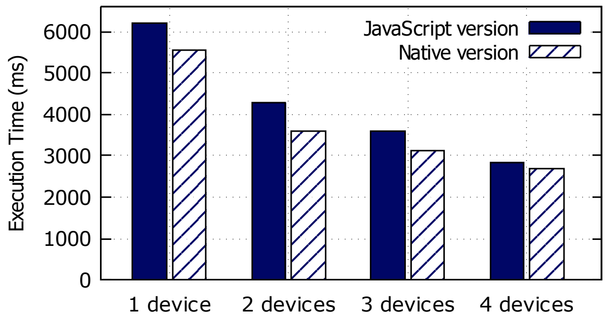


Fig. 5.12: Compare the performance of the JavaScript and Native versions in image processing test.

will sort the words by counting in a descending order, and then select and return the top n words. For this app, each job package's data will contain URL of the entire document, each peer will extract the content from the URL and process its portion. The result is in JSON format so the *Data Stripper and Aggregator* can merge the similar keys and increase the counts.

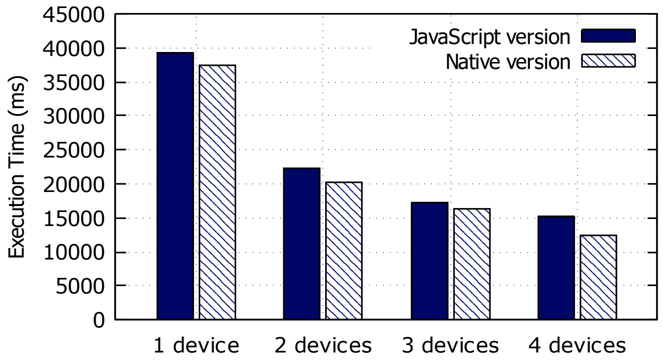


Fig. 5.13: Compare the performance of the JavaScript and Native versions with small data set.

Based on tests with a 1.5MB document, Figure 5.13 shows the similar trend as the previous case, especially running two devices are 45% faster than one on both versions. The difference of performance between the two versions are insignificant because the two processes only focus on fetching words without conversion. With 4 devices, the speed of both versions increase up to 65%.

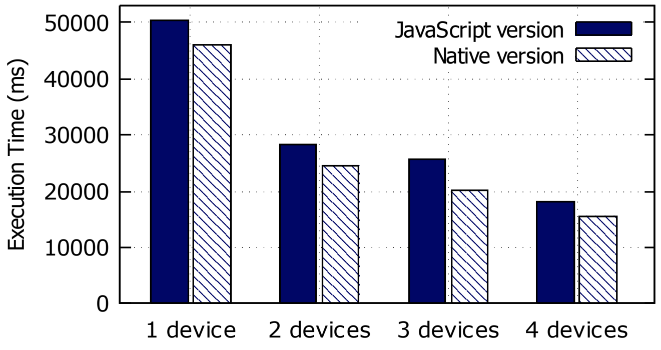


Fig. 5.14: Compare the performance of the JavaScript and Native versions with large data set.

On a 3MB document (Figure 5.14), we achieved the same performance increment from 44 – 64% for JavaScript version, and the total time of 2 devices running JavaScript version is 38% better than one running native version.

Tests with this sample app confirms that, in some cases, our JavaScript-based system can perform comparably with the native implementations.

PDF-to-Image Service

PDF-to-Image converting service is a typical example of loading an external JavaScript file dynamically. This kind of service could be used in a light PDF reader application for low-end devices with the remote rendering engine located on cloud server or remote devices. Since the *Loader Package* including a HTML and several JS scripts preexisted and remained unchanged on the remote device, we have to use `$.getScript()` from JQuery to load any external JS files (Code Snippet 5.3). Inside the callback `function()`, the JavaScript file is loaded and ready to use.

To load a PDF file, we use `PDF.js` library [19] from Mozilla which is a built-in module for PDF viewer on Firefox version 19+. The PDF converting service is straightforward, the requesting device wrap up an *App Request* which comprises of a PDF.js, PDF sample and information. When *Package Manager* receives this request, it will divide the request into a number of *Job Requests*, each will contain a few page indexes so that each remote devices will render a few pages of the total PDF document. (Code Snippet 5.3)

```
$.getScript('pdf.js', function() {
  PDFJS.workerSrc = './pdf.worker.js';
  PDFJS.getDocument('test.pdf').then(function(pdf) {
    /* load a page by page index */
        pdf.getPage(pageIndex).then(function(page) {
      var renderer = page.render(renderContext);
      ...
      /* check when the rendering process is done */
      renderer.then(function() {
        var imgData = $context.getImageData(0, 0,
                          viewport.width, viewport.height);
   ...
});
```

Code Snippet 5.3: Loading an external JavaScript file using JQuery.

We simplify the experiment by storing the PDF.js file on all peers because the requesting device doesn't need to resend the script file for every request. Figure 5.15 shows there is almost no benefits of running PDF converting service on multiple devices (1 to 5) in terms of the performance; it could probably gain when there is 8 more devices according to our estimation. Despite of the low performance, it is still a typical example of loading external JS scripts remotely and very useful for developing apps on low-end devices.

Stationary Server As A Peer

To verify the advantages of the system in comparison with the traditional offloading

[19] PDF.js: https://mozilla.github.io/pdf.js

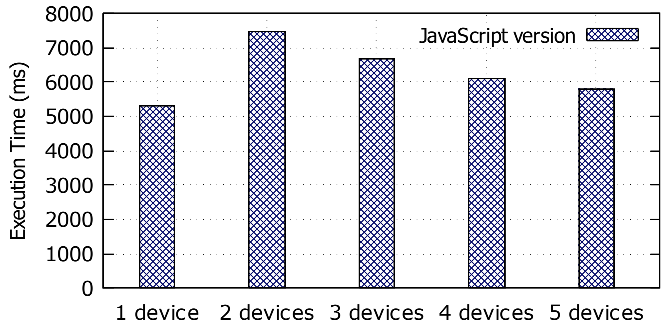


Fig. 5.15: Performance of PDF-2-Image converting service with 1 to 5 devices.

technique which relies on stationary server, we install Android x86 [20] on our test-bed server to mimic the server as a peer. Since the network bandwidth significantly affects the system performance, we use Microsoft Network Emulator [21] to virtualize three different networks LAN, WLAN and WAN and dispatch a number of different size packages on each network configuration.

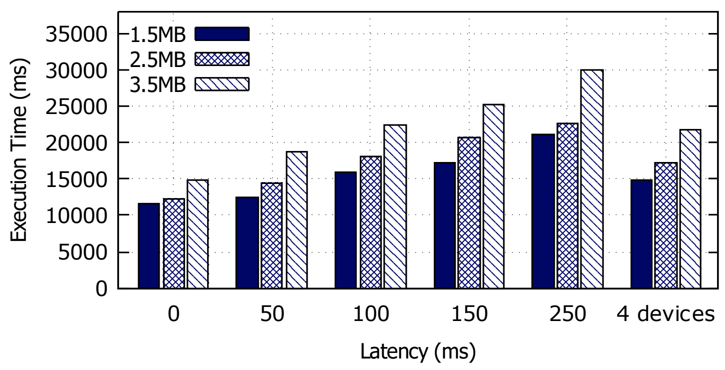


Fig. 5.16: Compare performance of server offloading and P2P networks with 4 devices.

According to the Figure 5.16, our JavaScript-based system on 4-device collaboration gives the same speed as offloading to the server at 100ms latency. At the lower latencies, the stationary server performs faster, at 21-32% on LAN (no latency) and 15% on WLAN (50ms latency) networks.

[20]Android x86: http://www.android-x86.org/

[21]Microsoft Network Emulator: https://goo.gl/j4kmc2

5.6 Related Work

With respect to using JavaScript for remote executions, Migratom.js [24] is a code migration system that uses a flow-based paradigm to accelerate mobile web applications by offloading compute-intensive tasks to the superpower servers. JSCloud [9] invokes the code analysis and instrumentation phases to decide whether to any offloading and which code partitions to offload to the cloud. JSCloud supports a wide range of devices and computers, but the estimation relying on interpolation incurs more computation cost. The other approaches [25,103,104] analyze offloadable code of JavaScript to enhance performance of web applications. Our *JSReX* differs from the other approaches by adopting WiFi-Direct to enable JavaScript-based collaborations over Device-to-device networks.

In code offloading, MAUI [4] introduces a code migration architecture relying on both *remote execution* and virtual machine migration to maximizes the potential for energy savings through fine-grained code offload while minimizing the changes required to applications. COMET [3] relies on Distributed Shared Memory (DSM) [105] to enhance VM-synchronization between the mobile devices and server for code offloading. This approach can support applications that contain no offloading logic, full multithreading, thread migration at any point during its execution and more efficient data movement. CloneCloud [97] uses a combination of static analysis and dynamic profiling to partition applications automatically at a fine granularity to enable unmodified mobile applications running in an application-level virtual machine to seamlessly off-load part of their execution from mobile devices onto device clones operating in a computational cloud. ThinkAir [90] proposes a different approach using smartphone virtualization in the cloud and parallelizing method execution for computation offloading. Our research extends the idea of these popular approaches to provide a solution for code offloading on multiple platforms using JavaScript-based execution packages.

Our research shares the common aspect with code offloading for mobile web applications. AppMobiCloud [106] employs a combination of profiling and points-to analysis at the development phase to find the computation-intensive code fragments, and specifies whether they can be offloaded with some constraints. Then, at runtime, it migrates the chosen JavaScript code fragments from the mobile devices for remote execution. It synchronizes client-side application runtime context and constructs the *cloned* context at server, executing the codes there and re-integrating the result back to the mobile device. Maciej et al. [107] introduces a new mechanism for web applications by offloading HTML5 web workers [22] from mobile device to the cloud. By extending the mobile web browser engine with offloading decision module, the system can decide whether to offload a JS background worker to the server farm. Similarly, Jeong et al. [108] apply a snapshot technique to safely save and restore the states of web applications and offload to server. This approach unties developer from any code limitations and constraints but it generates more overheads with server. In general, these approaches strongly bind to web applications which insist a stable communication for data exchanges with cloud server, they are inapplicable for device-to-device collaborations.

There are similar works that utilize WiFi-Direct technology but for the different purposes. Rio [43] leverages system I/Os to capture and share contents and resources between the existing applications running on different devices without any modification. Some of their applications are multi-system photography and gaming, singular SIM card for multi-devices, music and video sharing. GameOn, Prime [44, 109] are also built on the same networks to enable non-Internet connection between gamers within closed range areas like in public transportation.

Another relevant work is one that migrates different code bases into a system [73, 110]. Similar to *JSReX*, code migration mechanisms can execute code in different memory spaces like running C++ code on multi-core systems, using annotations to construct a privacy preserving data centric programming model [111] or thread migrations [112, 113].

5.7 Conclusion

This chapter has presented an initial version of *JSReX* – a novel distribution mechanism for offloading computationally intensive algorithms to peer devices in heterogeneous

[22]Web Worker: `https://w3c.github.io/workers/`

P2P networks. *JSReX* includes a simple distribution package and coding template so that developer can easily integrate offloading into their mobile apps. A performance evaluation of *JSReX* shows that it has low overhead for Android and Window Phone devices. The results of this initial evaluation are sufficient to motivate future research and development. Specifically, we will focus on (1) supporting native implementation for execution files, (2) privacy protection, (3) multi-group P2P networks, and (4) evaluations that examine other software qualities besides performance.

CHAPTER 6
INGRIM: A MIDDLEWARE TO ENABLE REMOTE METHOD INVOCATION ROUTING IN MULTIPLE GROUP DEVICE-TO-DEVICE NETWORKS

6.1 Abstract

Mobile devices can improve performance and preserve energy by offloading computational intensive calculations to nearby peers, as well as Internet-access servers. However, despite a long research history, peer-to-peer offloading is dilatory and unfit for applications which require rapidly consecutive requests over short periods. Existing solutions for inter-process communications are either unsupported or unwieldy for mobile platforms, or require Internet connectivity to access message servers or object brokers. This chapter introduces INGRIM– Inter-group Remote Invocation Middleware, a library-based middleware system to enable routing remote method invocation over multiple group device-to-device networks. INGRIM provides annotations for declaring distribution decision and out-of-box components that enable peer-to-peer offloading, even when a client app and the service provider do not have a direct network link or Internet connectivity. This chapter shows that INGRIM's overhead is similar to RMI, but that it can support inter-group communications.

6.2 Introduction

Mobile app developers have been offloading computationally intensive operations to servers for many years [3, 114], but the means for efficiently, effectively, and transparently offloading computations to nearby mobile devices (because servers are not always available or too costly) is still in its infancy. Although the core idea of peer-to-peer offloading is relatively straightforward and some frameworks for this purpose already exist [5, 86], there are open challenges that continue to hinder wide-spread offloading across mobile devices. These challenges span multiple areas, including: efficient task partitioning and distribution, improving developer productivity and software quality, and the seamless formation of ad hoc networks among the mobile devices.

Problems with task distribution deal with the selection of appropriate computations for offloading and their partitioning into distributable jobs, such that the resulting distributed computations have greater throughput or better energy efficiency [115]. To date, fully automated solutions in this area are limited and apply only to certain kinds of computational problems [86]. A more immediate and widely applicable approach is to give developers better tools for localizing [1] decisions about task partitioning and distribution, either imperatively or declaratively.

The second major area focuses on software engineering issues, including developer productivity and software quality. The demand for more sophisticate apps is only increasing. For example, end users are coming to expect ubiquitous inter-device connectivity, especially for social interactions; app responsiveness even for complex operations like those required for augmented reality; the advertised battery life for their devices regardless of which apps are running; and complete functionality of an app regardless of Internet connectivity. Consequentially, apps are becoming more complex and developers must find ways to manage that complexity while ensuring that their software is reliable, secure, maintainable, extensible, and efficient, as well as possessing any other properties considered necessary for quality in the app's domain [117]. Commonly, solutions in the areas take the form of libraries, frameworks, patterns, or middleware that provide developers with high-level abstractions for inter-process communications or object distribution. Examples include remote method innovations like Java's RMI, object brokers like CORBA [118], and messaging systems like JMS or RabbitMQ [66]. However, these solutions are either unavailable on mobile platforms, dependent on Internet connectivity to access specialized servers, or are too heavy weight for most mobile devices. See Section 2 for more details.

The third area deals with the application-level connectivity in situations where devices only have single-hop link-level connectivity to other devices within WiFi or bluetooth range,

[1]The localization of design decisions is an aspect of good modularization. It reduces code duplication and, when coupled with good encapsulating and abstraction, can greatly enhance the maintainability, extensibility, and reusability of a software system [116]

as is commonly the case when mobile devices are out of range of a WiFi access point or lack a cellular data connection. Communication subsystems, like WiFi Direct (see Section 2), support this kind of single-hop connectivity but not support connectivity across overlapping groups of devices, i.e., communications that would require multiple hops.

This chapter presents an innovative library-based middleware system, called INGRIM – a Inter-group Remote Invocation Middleware, that addresses at least one problem in each of the three areas mentioned above. Specifically, it aims to

- Make it easy for developers to declare which services can be offloaded and the optimal communication patterns for executing each service (Area 1).
- Support complete access and location transparency[2] [119] for individual services (Area 2).
- Enable the integration of offloading into existing, as well as new apps, with minimal effort (Area 2).
- Provide developers with high-level abstractions that hide all but the architectural-design details of inter-group communications (Areas 2 and 3).
- Perform comparably to RMI for single-hop inter-process communications (Area 3) and support multi-hop communications without traditional access points or routers, which RMI doesn't support (Area 3).

To meet these goals, INGRIM provides the following features:

- Annotations that allow developers to localize and encapsulate offloading decisions.
- Automated tools for generating service proxies and skeletons, plus the ability for developers to customize their functionality.
- Out-of-the box components for setting up and managing inter-group (multi-hop) communications.
- Out-of-the box components that automatically manage service locations and route requests/replies between clients and services

[2]With access and location transparency, a client component can call a service that may be offloaded at runtime without worrying about whether the service is actually executed locally or remotely and, if remotely, where the service is hosted.

- Connectivity with Internet-accessible services, in addition to those on peer devices
- Compatibility with mobile platforms that support a Java Virtual Machine (JVM) and WiFi Direct.

INGRIM's annotations and code generators provide the developer with easy-to-use tools for localizing and encapsulating distributions discussion in way supports access and location transparency to service clients. INGRIM's out-of-the-box components include a broker, which is responsible for location-transparency communications, and two kinds of bridges, which are responsible for inter-broker communications, especially between network groups. Developers can use the annotation, code generators, and out-of-the-box components to integrate peer-to-peer offloading into virtually any type of mobile application by (1) deciding what services (methods) can be offloaded and annotated appropriately, (2) customizing the proxy and skeleton code generated as a result of those annotations, as needed, (3) deciding the high-level architecture, i.e., which devices will have brokers and bridges, and then writing startup code to instantiate those objects along with application's service objects. Section 3 provides an architectural overall of INGRIM and describes its components in more detail.

Section 4 provides an assessment of INGRIM with respect to goals listed above, including the results of several experiments that were used to evaluate its performance. Section 5 discusses other related works. Finally, Section 6 discusses future work that will address other outstanding issues in each of the mentioned problem areas and summarizes INGRIM's contributions as a step forward in helping peer-to-peer offloading reach its full potential.

6.3 Background

The range of technologies that can and have been applied to distributed applications, and specifically offloading, is vast and diverse. To understand INGRIM and its contributions, though, it is only necessary to review a few representative technologies, specifically remote method invocations, object brokers, and messaging systems (see Section 2.1) and to provide a few details about WiFi Direct (see Section 2.2) and ZeroMQ (see Section 2.3).

We chose to build INGRIM on top of WiFi Direct and ZeroMQ for their small footprint, simply interfaces, and availability on common mobile platforms.

6.3.1 Representative Technologies

Java's RMI is a mechanism for invoking methods on remote objects [120]. One of its strengths is that it provides a degree of location transparency by having servers add services to a registry and requiring clients to use the registry for binding. But, it does not support any routing other than what the underlying network layers support. So, if there is no inter-network (network-layer routing) between two devices, an object on the first device cannot invoke a method for an object on the second device. Unfortunately, such inter-group communication is a common situation with mobile devices and one of the problems that INGRIM is trying to address.

Technological based on object brokers, like CORBA, also support remote method invocations. But, they rely on middleware processes (or threads) to instantiate or re-hydrate objects and then bind method calls to the target objects. Although there have been attempts to support CORBA on mobile platforms [121], they require the underlying layers to handle inter-networking and therefore do not directly support inter-group communications. Also, the authors believe that its middleware is too heavy for most mobile devices and that its language-neutral approach to distribution is unnecessarily complex for most mobile apps.

Messaging systems, like JMS, RabbitMQ, and ActiveMQ [80] can also play a role in offloading. Although they do not directly support remote method invocation, they provide reliable routing and flow control through queuing mechanisms. The problem with most of these technologies is that they are too unwieldy for mobile platforms and they typically depend on servers for message queuing. There is one lightweight and portable messaging library that doesn't require a server, and that is ZeroMQ. See Section 2.3.

6.3.2 WiFi Direct

WiFi Direct is a new peer-to-peer communication standard built on top of the IEEE

802.11 to provide direct connections between the Wi-Fi-enabled devices without Internet connections [122]. In our research [6,7], we rely on WiFi Direct to construct opportunistic networks, i.e., groups, among the nearby devices, by letting them dynamically discover and connect to each other. However, with WiFi Direct, a single device can only belong to a single group at any time. It is possible, though, for a device to still use its legacy WiFi client (LC) to connect to an Internet access points or any other peer device directly.

In terms of performance, it is important to note that WiFi Direct adds another layer on top the underlying communication layers, so messages sent through WiFi Direct are expected to have slightly higher overhead than message sent an LC. To discover the efficiency of WiFi Direct in opportunistic networks, we compare the performance of plain WiFi to WiFi Direct by sending and receiving messages of various sizes and measuring the turnaround time. Figure 6.1 shows the similarity in performance between the two protocols when sending messages from a mobile device [3] to a remote server and receiving a reply. The message sizes varies from 1KB to 1MB.

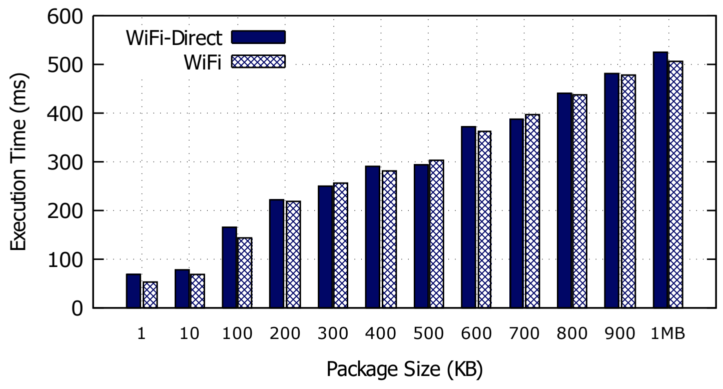


Fig. 6.1: Performance comparison of WiFi and WiFi Direct.

WiFi Direct forms short-range opportunistic networks by polling available nearby devices and electing a Group Owner (GO) [123]. When a device becomes GO, it establishes a virtual access point (i.e., soft AP) and starts a DHCP service to automatically assign IP addresses (range of 192.168.49.0/24) for itself and other clients of its group.

[3]Runs on Android-X86 platform with WiFi Adapter

Since WiFi Direct only allows each device to belong to one group, the members of a group (including the group owner) cannot use WiFi Direct to talk to members of another group, or to an Internet access point for that matter. Fortunately, WiFi Direct does not preclude the direct use an LC and it exposes a GO's soft AP to other devices outside the group. So, an app running in another group can use its LC to establish a communication link to the GO of the first group. After its LC is connected, that LC will be assigned an IP in from the GO's DHCP's IP range (See Figure 6.2). INGRIM and others, like Funai et al. [123], have exploited this technique to support inter-group and group-to-Internet communications when using WiFi Direct. A similar approach has been successfully applied in the content-centric routing domain [29].

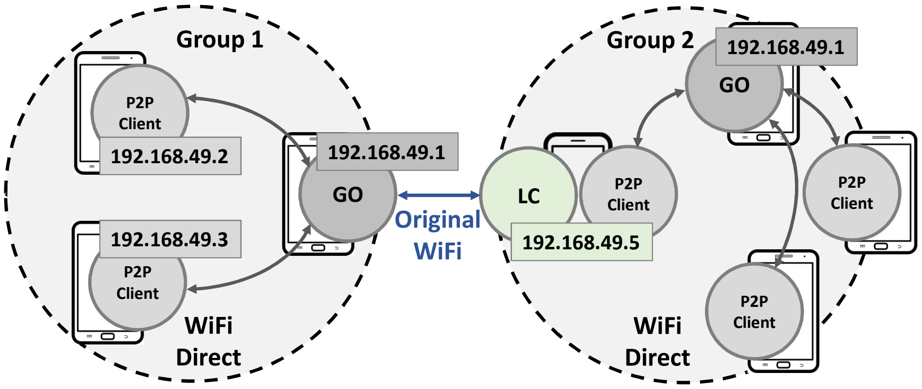


Fig. 6.2: Use of original WiFi interface to create group-to-group communications.

6.3.3 ZeroMQ

ZeroMQ (ZMQ) [79] is a flexible messaging library with native implementations in C/C++, Java, and C# and with bindings for 40+ other languages [4]. It is lightweight and therefore well suited for mobile apps.

ZeroMQ brings to developer sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. Developer can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast

[4]ZeroMQ: zeromq.com

enough to be the fabric for clustered products. Its asynchronous I/O model enables scalable multicore applications, built as asynchronous message-processing tasks.

6.4 INGRIM: Inter-group Remote Invocation Middleware

As mentioned earlier, with INGRIM, developers integrate offloading into mobile apps by (1) annotating services, (2) customizing the proxy and skeleton code, (3) deciding the high-level architectures and writing startup code to instantiate the necessary brokers and bridges.

Employing the middleware is quite simple via 2 steps: (1) developer creates service with full implementation and marks the methods with service annotations (Code Snippet 6.1). After compiling the entire project, the INGRIM's processor will automatically generate extension classes for the service. Then (2) developer will use these generated classes along with the other basic components such as `Broker` and `Bridge` to construct their mobile networks (An example is in the Code Snippet 6.5).

6.4.1 Service Definition

Developer indicates a class as service by declaring `@MobileService` annotation on the class prototype, where *transmission type* can be either binary (`TransmitType.Binary`) or JSON format (`TransmitType.JSON`). By default *transmission type* is set to `Binary` but user can select `JSON` for the simple request wrapping primitive data such as *String*, this option is useful to display messages while they are routing in `DEBUG` mode. In the entire Section 6.4, we only discuss the `Binary` option.

Service functions always come with `@ServiceMethod` annotation, the other functions without this annotation will be excluded during the compilation. There are two options for function *synchronous mode*: `Async` - the result is handled by a *common handler* and system can jump immediately to the next call, `Sync` - the request device is blocked until the result arrives. Code Snippet 6.1 shows a service sample with two simple functions `greeting` in `Sync` mode and `getFolderList` in `Async` mode.

```
@MobileService(transmitType = TransmitType.Binary)
public class ServiceA {
    @ServiceMethod(syncMode = SyncMode.Sync)
    public String greeting(String msg) {
        return "Hello " + msg;
    }
    @ServiceMethod(syncMode = SyncMode.Async)
    public String[] getFolderList(String path) {
        File folder = new File(path);
        File[] files = folder.listFiles();
        String[] res = new String[files.length];
        for (int i = 0; i < files.length; i++)
            res[i] = files[i].getAbsolutePath();
        return res;
    }
}
```

Code Snippet 6.1: Service definition example.

During the compilation, the `AnnotationProcessor` module will automatically generate the *Proxy* and *Skeleton* classes that are going to be placed on the local and remote devices. *Proxy* is a generated class which resides on the local device to dispatch function call requests. It has the same function list as the Service but inner implementation is the generated code to convert function call to a request message (Code Snippet 6.2). A *Proxy* contains an instance of `FrontEnd` for *synchronous* calls and a `Sender` for the *asynchronous*, for `ServiceA`, the *Proxy* has assigned name by default, which is `ServiceAProxy`.

The *Skeleton* is a generated class which resides on the remote device to resolve function requests. *Skeleton* inherits `BackEnd` class and contains an instance of the Service to call the corresponding function if it receives a request with the same `functionName`. By default, the Skeleton is assigned with the name `ServiceASkeleton` (Code Snippet 6.3).

6.4.2 Proxy and Skeleton In Use

INGRIM library is modularized by components that are possible to cooperate with both WiFi and WiFi Direct, application developer therefore can opt to construct any network

```
public class ServiceAProxy {
  FrontEnd frontend;
  Sender sender;

  public String greeting(String msg) {
    /* generated code */
  }
  public void getFolderList(String path) {
    /* generated code */
  }
}
```

Code Snippet 6.2: Generated Proxy class.

topology they may expect. In this section we describe a very simple form of network we could make using the Proxy and Skeleton classes from the prior section (Figure 6.3), the more complicated architecture can be varied (Figure 6.9).

```
public class ServiceASkeleton extends BackEnd {
  ServiceA service;

  @Override
  public byte[] resolveRequest(byte[] reqBytes) {
    byte[] respBytes = null;
    RequestMessage reqMsg = NetUtils.deserialize(reqBytes);

    switch (reqMsg.functionName) {
    case "greeting": {
      /* generated code */
    }
    case "getFolderList": {
      /* generated code */
    }}
    return respBytes;
  }
}
```

Code Snippet 6.3: Generated Skeleton class.

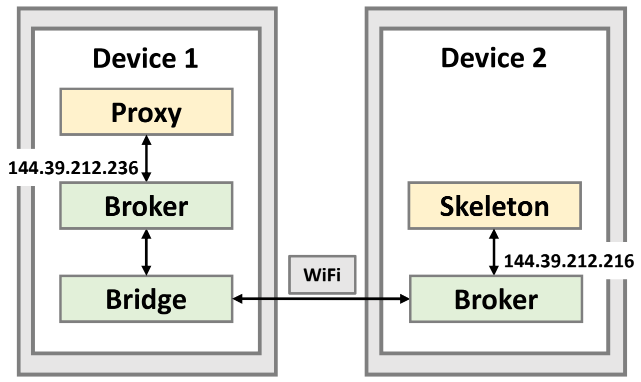


Fig. 6.3: Simple use of the Proxy and Skeleton objects.

According to the design in Figure 6.3, we firstly install the `ServiceASkeleton` on the remote device, it will come with a Broker to host the services and trigger connection with another device (Code Snippet 6.4).

```
/* start a Broker and a BackEnd on the remote device */
new Broker(remoteBrokerIp, clientPort, workerPort);
new ServiceASkeleton(remoteBrokerIp, workerPort);
```

Code Snippet 6.4: Example code to run Broker and Skeleton on the remote device.

On the local device we start a Broker to host services and a Bridge to reach out for the remote Broker. Then, we start `ServiceAProxy` and declare how to handle incoming responses from *asynchronous function calls* in the `receive()` callback. In the callback implementation, `msgType` indicates whether a message is *information* (`BROKER_INFO`) or *response message* (function name), the developer is required to add appropriate code to manipulate the responses (Code Snippet 6.5).

6.4.3 INGRIM Components

The middleware is constituted by 6 main components: `Broker`, `BackEnd`, `FrontEnd`, `Sender`, `Receiver` and `Bridge`, each has different functionality but shares the basic structure including *ring buffers* to buffer incoming and outgoing messages, and *ZMQ sockets*.

```
/* start a Broker and a Skeleton objects on local device */
new Broker(localBrokerIp, proxyPort, skelPort);
/* start a Bridge to bridge the local and remote Brokers */
new Bridge(localBrokerIp, skelPort, remoteBrokerIp, proxyPort);
/* start a Proxy at local */
ServiceAProxy proxy = new ServiceAProxy(localBrokerIp,
                            proxyPort, new ReceiveListener() {
  @Override
  public void received(String IDs, String msgType, byte[] data){
    ResponseMessage resp = NetUtils.deserialize(data);
    if (msgType.equals(NetUtils.BROKER_INFO)){
      /* a denied message from the Broker */
      Log.v("Error: " + resp.outParam.values[0]);
    }else if (msgType.equals("getFolderList")){
      /* results from the "getFolderList" function */
      String[] files = (String[]) resp.outParam.values;
      for (int i = 0; i < files.length; i++)
        Log.v("File: " + files[i]);
    }
}});
/* call the function from the proxy */
proxy.getFolderList("/");
```

Code Snippet 6.5: Example code to run Broker, Bridge and a Proxy on the local device.

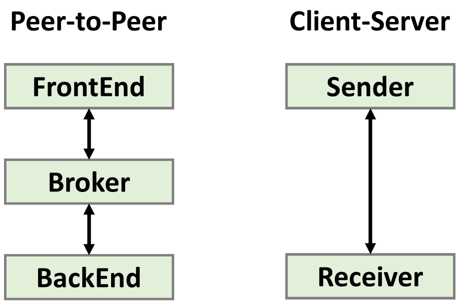


Fig. 6.4: INGRIM components – Broker, FrontEnd, BackEnd, Sender and Receiver.

Figure 6.4 and 6.5 depict the preexisted communications among the components. In Figure 6.4-Left, `BackEnd` connects and registers its services to `Broker` while the `Broker` buffers the request messages sent from `FrontEnd`, forwards each request to the corresponding `BackEnd` to resolve and forwards result back to the `FrontEnd`. This type operates in *asynchronous* mode, the `FrontEnd`'s callback handler is where incoming messages such as results and info messages are proceeded. Figure 6.5-Left introduces a more sophisticated strategy with the involvement of a `Bridge`, an intermediate between two `Brokers`. The `Bridge` consists of a `FrontEnd` and `BackEnd`, one connects to the left `Broker` and another connects to the right. These two types will be used for *Peer-to-Peer* and *Group-to-Group* modes.

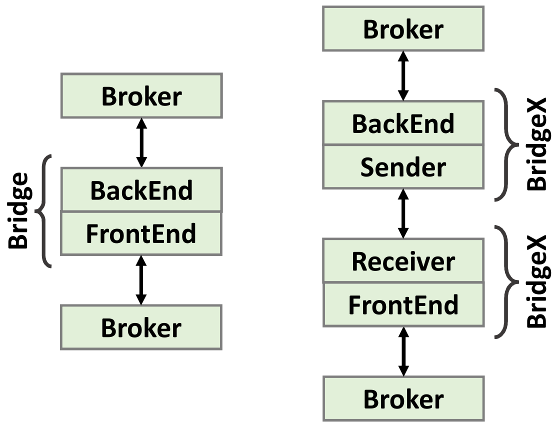


Fig. 6.5: INGRIM components – Bridge and BridgeX.

Figure 6.4-Right depicts `Sender` and `Receiver`, one sends request and the other responses in *synchronous* mode. An advantage of this type is no Broker involved, making the message routing time becomes shorter, so that it could be perfectly employed in *Client-Server* model. However, since this type works in *synchronous* mode, in our first implementation we only used it for `BridgeX`. Figure 6.5-Right shows the constitution of `BridgeX` which consists of 4 components `FrontEnd`, `Sender`, `Receiver` and `BackEnd`, in the design, two of sub components contact the left `Broker` and the other twos contact the right. The benefit of `BridgeX` is that they can connect to each other directly without a middle `Broker`, while `Bridges` insists one (Figures 6.9 and 6.10).

These components don't start at the same time. Generally, `Broker` always starts first right after network has been established to either host services for its current group or interconnect with the other groups. Then, `BackEnds` come after `Broker` to register their

```
"action":"REGISTER",
"id":"1",
"functions":[
  { "functionName":"greeting",
    "inParams":["String"],
    "outParam":"String[]"},
  { "functionName":"getFolderList",
    "inParams":["String"],
    "outParam":"String[]"}]
```

Code Snippet 6.6: BackEnd's service definition in JSON format.

services, when it starts, it sends service definition in JSON format (Code Snippet 6.6) to the `Broker`. The service definition consists of (1) `action` as an indicator for `Broker` to register/unregister the `BackEnd`'s service, (2) BackEnd's `ID` and (3) `functions` – the list of provided functions, each contains information of function name, input and output parameters.

`Broker` will extract the *function list* and store them in `FunctionMap` table where *keys* are *function names* and *values* are *BackEnd IDs*. Later, the `Broker` will use `FuncName` from a request message to find the according `BackEnd` and forward the request to it. Before `BackEnd` leaves the network, it sends the `Broker` an instruction message with code `UNREGISTER` to remove all of its functions out of the `Broker`'s `FunctionMap`.

`Bridge` is simply a forwarder which starts when the `BackEnd`s have been settled. At the beginning, it sends a *Service Request* to the remote `Broker` for a list of available remote functions, the `Broker` will response by returning the function list in the same format as the `functions` in the Code Snippet 6.1. Then, it will connect to the local `Broker` and register those remote functions, the local `Broker` will register those functions under the `Bridge`'s ID.

In our implementation, only `Broker` and `Bridge` are used in their original forms, the other components will be generated according to the developer's service declaration during the code compilation.

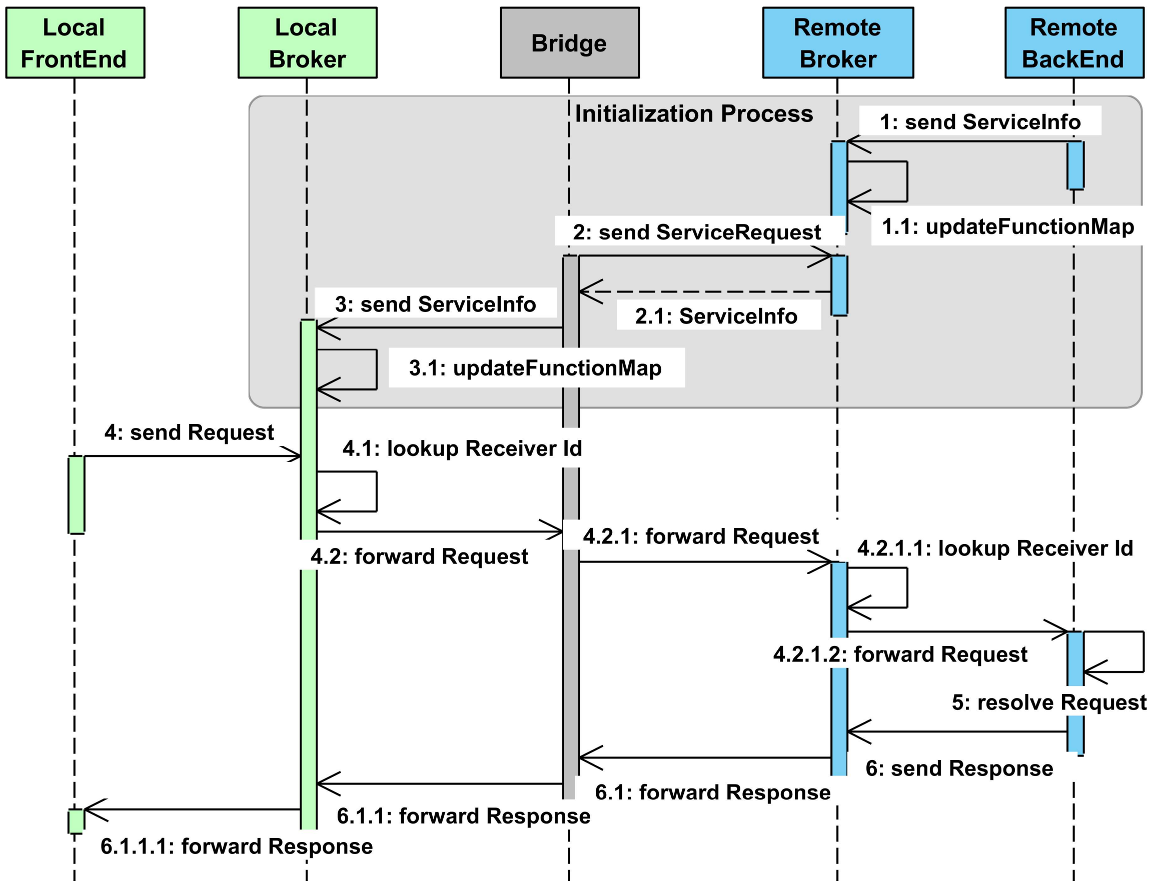


Fig. 6.6: Sequences of the initialization process and message requests.

6.4.4 Function Call To Message Conversion

On FrontEnd/Sender, INGRIM dispatches a function call over the network by converting it into a *request message* object, then serializing the request into binary array for network transmission (Code Snippet 6.7). On BackEnd/Receiver, it deserializes the binary array back to *request message* object, passes the object data into parameters and calls the real function. Finally, it fills the result into a *response message*, serializes into binary format and sends to the network (Code Snippet 6.8). To this end, `AnnotationProcessor` module scans the entire project and generates the wrapping classes including the pairs: FrontEnd–BackEnd and Sender–Receiver for all defined services marked with `@MobileService` annotations.

On FrontEnd/Sender, the `RequestMessage` class defines the *request message* object.

The request includes: (1) `functionName` to keep the name of function, (2) `InParams` to contain types and values of input parameters and (3) `OutParam` to describe the type of output parameter; the parameter type can be a single value or an array of any primitive or user-defined object, as long as the relative classes exist in the *classpath* during the compilation and execution on all devices. The Code Snippet 6.7 shows how the function `getFolderList` with one parameter (Defined in the Code Snippet 6.1) is transformed into a `RequestMessage` object and serialized into binary array.

```
public String greeting(String msg) {
  // create a request message & serialize it to binary format
  String functionName = "greeting";
  String outType = "String";
  RequestMessage reqMsg = new RequestMessage(
                              functionName, outType);
  reqMsg.inParams = new InParam[1];
  reqMsg.inParams[0] = new InParam("msg", "String", msg);
  byte[] reqBytes = NetUtils.serialize(reqMsg);
  // send request to network & halt until response is available
  byte[] respBytes = req.send(functionName, reqBytes);
  /* deserialize the response and extract the output */
  ResponseMessage resp = NetUtils.deserialize(respBytes);
  String output = (String) resp.outParam.values[0];
  return output;
}

public void getFolderList(String path) {
  String functionName = "getFolderList";
  String outType = "String[]";
  RequestMessage reqMsg = new RequestMessage(
                              functionName, outType);
  reqMsg.inParams = new InParam[1];
  reqMsg.inParams[0] = new InParam("path", "String", path);
  byte[] reqBytes = NetUtils.serialize(reqMsg);
  frontend.send(functionName, reqBytes);
}
```

Code Snippet 6.7: Generated code in Service Proxy class.

Code Snippet 6.8 shows the execution mechanism of BackEnd/Receiver to handle a request by deserializing the binary data back to `RequestMessage` object and categorizing it using `functionName` attribute. Inside each *method handler* (each *case*), input parameters collected from `InParams` attribute of the request message are passed to the actual function call of the service instance (`serviceA`) and the output type is from `OutParam`. Finally, the result of the call is wrapped within a `ResponseMessage` along with the name and type, then it is thrown back to the Broker (Code Snippet 6.8). To support asynchronicity, BackEnd or Receiver handles each request on a single thread, the middleware allow at most 5 threads running simultaneously by default.

6.4.5 Message Flows

In this section, we describe the design of low-level message flows on top of ZMQ from FrontEnd to BackEnd (the message flow from Sender to Receiver is almost the same) through a few Brokers and Bridges and vise versa.

In ZMQ, a message traveling between the two sockets needs at least 2 parameters: *identity* of the destination and *message content*. To avoid overheads of message transit on the intermediates, we design message format with the following fields: `ReceiverId` – identity of the destination, `IDs` – ID chain of passed FrontEnds on the route, `FuncName` and `Message` – a binary form of serialized `Message` object. Particularly, `IDs` keeps a series of FrontEnd IDs which it passes along the way to BackEnd, for example in Figure 6.7 when the message arrives at the BackEnd the value of `IDs` is ”`1/100/200`” where 1 is the ID of FrontEnd #1, 100 is the ID of Bridge's FrontEnd #1 and 200 is the ID of Bridge's FrontEnd #2. `IDs` is concatenated when the message arrives at Broker from the FrontEnd and popped out to use when it arrives at the Broker from BackEnd. Finally, a message comes with `startTime` and `timeout` to define how long the message should be available in the route, the request will be marked as *failed* if the response doesn't come out before the timeout (Sub Section 6.4.7).

```
/* create an instance of actual service here */
ServiceA service = new ServiceA();
RequestMessage reqMsg = NetUtils.deserialize(packageBytes);

switch (reqMsg.functionName) {
case "greeting": {
  /* variable "msg" */
  String[] msgs = new String[req.inParams[0].values.length];
  for (int i = 0; i < req.inParams[0].values.length; i++)
    msgs[i] = (String) req.inParams[0].values[i];
  String msg = msgs[0];
  /* start calling function "greeting" */
  String[] rets = service.greeting(msgs);
  String retType = "String";
  ResponseMessage resp = new ResponseMessage(req.messageId,
                             req.functionName, retType, rets);
  /* convert to binary array */
  return NetUtils.serialize(resp);
}
case "getFolderList": {
  /* variable "path" */
  String[] paths = new String[req.inParams[0].values.length];
  for (int i = 0; i < req.inParams[0].values.length; i++)
    paths[i] = (String) req.inParams[0].values[i];
  String path = paths[0];
  /* start calling function "getFolderList" */
  String[] rets = service.getFolderList(path);
  String retType = "String[]";
  ResponseMessage resp = new ResponseMessage(req.messageId,
                             req.functionName, retType, rets);
  /* convert to binary array */
  return NetUtils.serialize(resp);
}
```

Code Snippet 6.8: Generated code in Service Skeleton class.

Sending A Request

We describe the Request flow using a typical example in Figure 6.7: A message to Broker doesn't need an address because a FrontEnd connects to only one Broker, so the first message's `ReceiverId` is `EMPTY` and `IDs` is "`1`" since the message came out from FrontEnd with ID is 1. When Broker receives the request, it looks up `FuncName` in the `FunctionMap` to find the corresponding BackEnd/Bridge and forwards the message, for this example the destination is a Bridge. The Bridge concatenates `IDs` with the ID of its FrontEnd ("`1/100`" - since Bridge's FrontEnd ID is 100) and forwards the request to the next Broker. This process repeats until the request eventually meets the BackEnd which owns the requesting function and gets resolved.

If for any reasons the request can't find the BackEnd, a denial message with flag `BACKEND_NOT_FOUND` will be sent back to the FrontEnd as response. This case happens when the request message gets lost at a Broker where the requesting function is not available in its list.

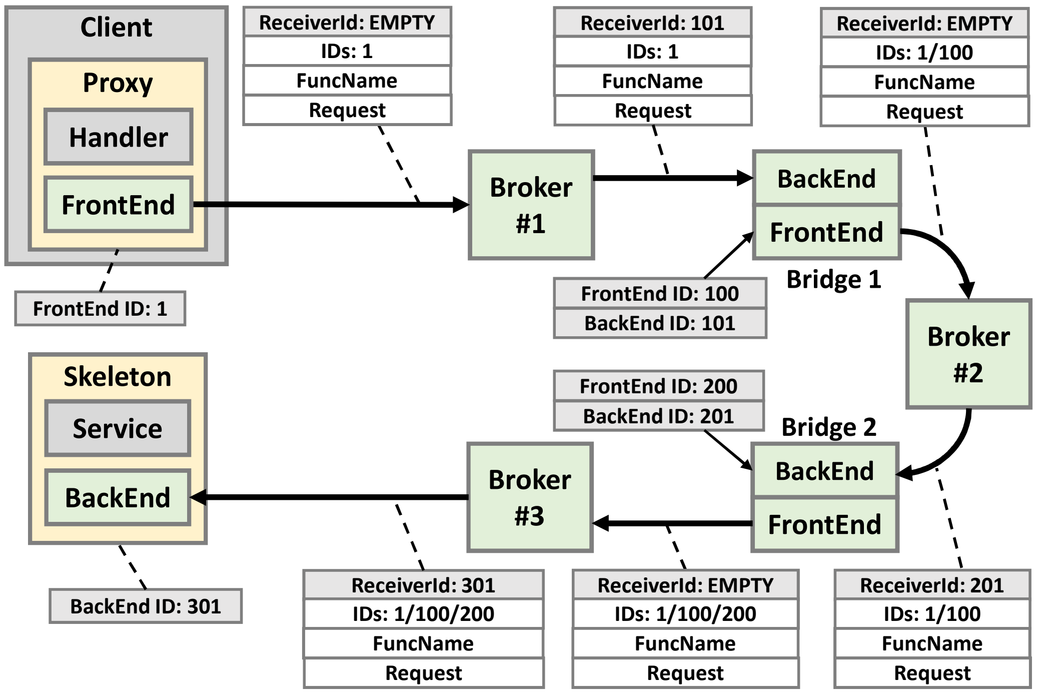


Fig. 6.7: Message flow from a Service Proxy to the remote BackEnd.

Sending A Response

Figure 6.8 illustrates a Response flow from BackEnd to the requesting FrontEnd. When the response arrives at Broker, the Broker will extract the first ID in the `IDs` and put it to the `ReceiverId` so that the response can find way to the next FrontEnd. If the FrontEnd has not defined handler for the response, it will forward the message to the next Broker. This process repeats until the `IDs` is `EMPTY`, in other words the response arrives at the requesting FrontEnd.

If the response can't find the way back to the FrontEnd (when `ReceiverId` not found or `IDs` is `EMPTY`), the FrontEnd will wait until timeout to report `UNAVAILABLE_SERVICE` error.

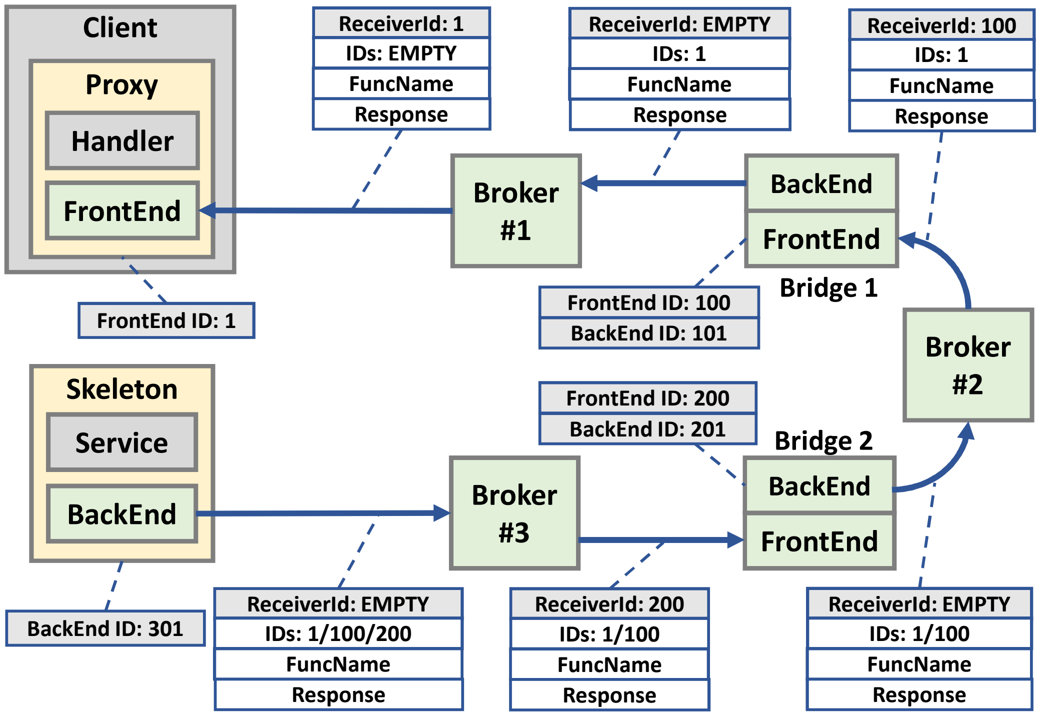


Fig. 6.8: Message flow from a BackEnd back to the FrontEnd.

6.4.6 Group-to-Group Communications

This section will go deep inside the deployment of INGRIM to achieve the goal. Figure 6.9 illustrates a typical case of two groups 1 and 2, each group has 2 devices: one takes role of GO with a Broker (host on the default WiFi-Direct IP – 192.168.49.1) and another starts a FrontEnd.

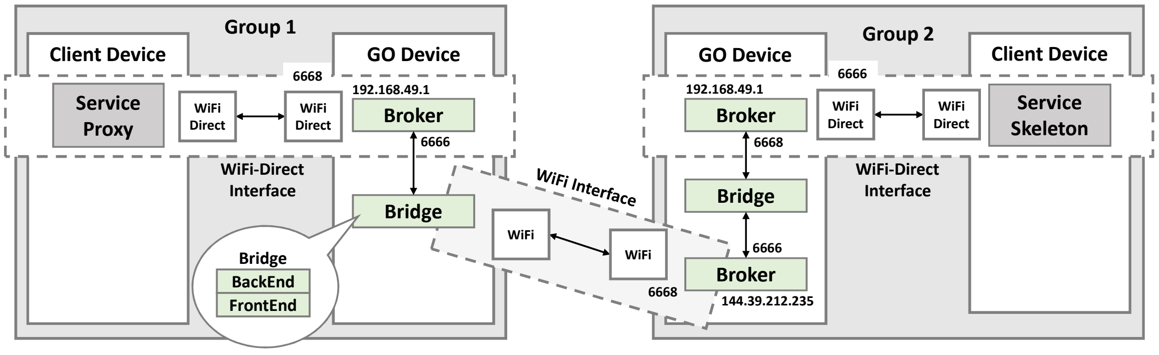


Fig. 6.9: Architecture of multi-group device-to-device communication with 4 devices.

To implement a *Legacy Client*, we firstly let the Group 1's GO connect to the WiFi AP created by Group 2's GO [5]. Then, on the Group 2's GO, we start a new Broker to host on the IP address of the WiFi interface, for instance 144.39.212.235, which is completely irrelevant to the WiFi-Direct network established before. A new Bridge will start on the Group 2's GO to connect the two Brokers; Likewise, a new Bridge will start on the Group 1's GO to connect its Broker with the new Broker on Group 2's GO over the WiFi. From this moment, the system will operate exactly the same way as the one we described in the Section 6.4.5.

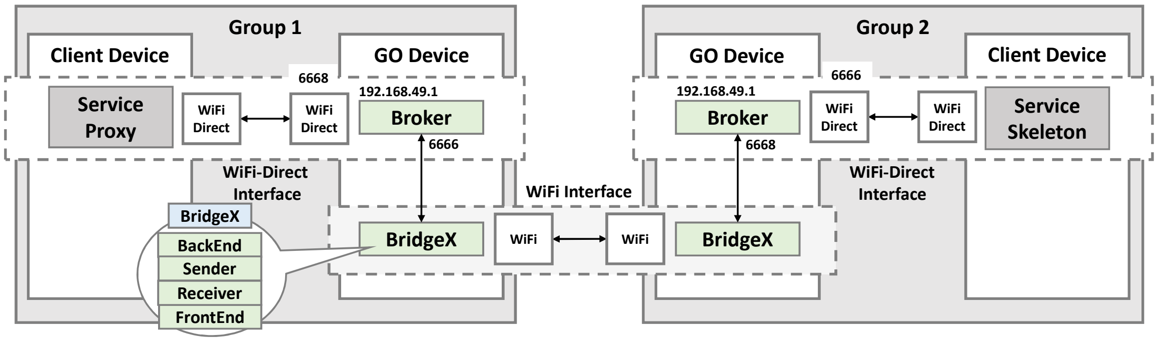


Fig. 6.10: Architecture of multi-group device-to-device communication with 4 devices.

[5]When a device becomes GO, it will also be a WiFi Access Point. The other devices can connect to that AP via the WiFi interface.

To simplify the complexity of system, we use `BridgeX` to replace one Broker on Group 2's GO device (Figure 6.10).

As aforementioned, INGRIM can work on both Android and PC platforms, developer can easily bridge a communication from a device to PC by deploying a Broker on PC to host the connections from devices (Figure 6.15). On mobile device, we could use Bridge to relay messages back and forth from mobile Brokers to PC.

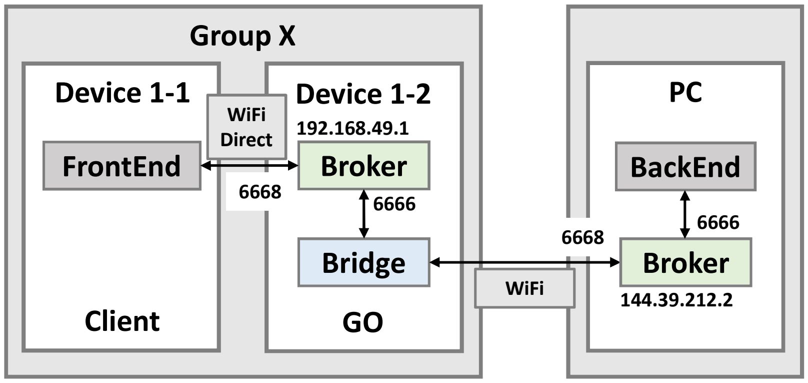


Fig. 6.11: A simple way to bridge device to PC.

6.4.7 Failure Handling

INGRIM guarantees the client will receive either final result or appropriate error message of any malfunction happens while routing.

On the requesting device, the FrontEnd maintains a copy of each request and it frequently checks up in every 500ms on the request list. If a request has passed the `timeout`, the FrontEnd will stop the listener thread for that request, report an `UNAVAILABLE_SERVICE` message and opt to execute it locally.

On Broker, if a forwarded request to BackEnd has passed the network timeout, the Broker will attempt sending it again. If the failure still occurs, the Broker will consider the BackEnd is unreachable, then it closes the sending thread and remove all the relevant service functions of the BackEnd out of its list. Then, it will throw `BACKEND_UNREACHABLE` error message back to the requesting device. Sometimes later, if the Broker receives a request with `funcName` is unfound in its function list, it will throw `BACKEND_NOT_FOUND` back to the requesting device.

When Broker receives a request or response, it finds the subtraction between the current time and message's `startTime`, then compares with `timeout`. If the message passes the `timeout`, the Broker will replace the current message by an `TIMEOUT` error message and send it back to the requesting device using the current message's `IDs`.

On BackEnd, if it cannot send back a response within the network timeout, it will make the second attempt. If failure still occurs, the BackEnd will consider the Broker is unreachable, then it disconnects the Broker and close the sending thread. When it actively leaves the group, it will send a `UNREGISTER` message to its Broker to let it remove all the relevant service functions from the list.

6.5 Evaluation

We built a test-bed with actual WiFi-Direct featured Android devices to evaluate the performance of our system, a PC is also included to examine the bridge between mobile devices and stationary computer (Table 6.1).

Table 6.1: List of devices used in the experiments in chapter 6.

Device	**CPU**	**RAM**	**Battery**
LG Volt	Quad-core 1.2GHz	1GB	3000mAh
ZTE Maven 3	Quad-core 1.1GHz	1GB	2115mAh
Moto G4	Octa-core 1.5GHz	2GB	3000mAh
BLU R1	Quad-core 1.3GHz	2GB	2500mAh
Lumia 550	Quad-core 1.1GHz	1GB	2100mAh
Dell PC	Intel i7-4790 3.6GHz	8GB	Wall-plugged

6.5.1 Micro Benchmark

We designed a simple service with one function accepting a binary array as input parameter and returning size of the array. A component (e.g. FrontEnd), when forwarding a function call, will pack and send the function message with parameter values out to another one. In this section, we will gradually increase size of the binary array from 1KB to 1MB in order to figure the performance of the components with and without network.

The measure time $T_{[Total]}$ will be estimated at the FrontEnd following the Equation 6.1, with $T_{[Net]}$ is the total network round-trip time of all components.

$$T_{[Total]} = T_{[Broker]} + T_{[Bridge]} + T_{[BackEnd]} + T_{[Net]} \tag{6.1}$$

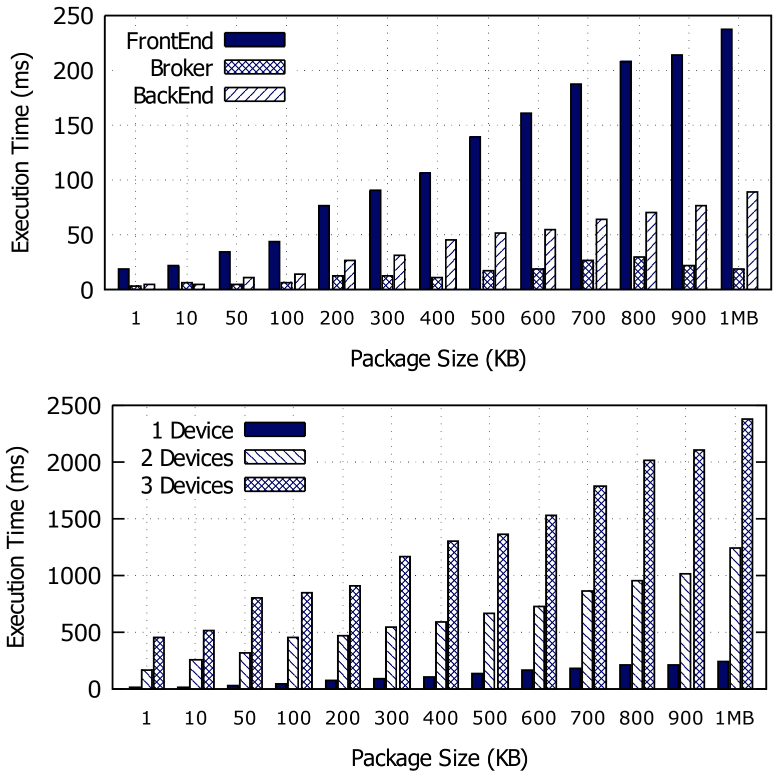


Fig. 6.12: Benchmark tests by running all components on one device.

For overhead measurement of each component, we isolate the network usage by running all components on one single device. Figure 6.12-1 shows the promising result where Broker only spends 5-30ms to store and forward a request while FrontEnd and BackEnd steadily increase processing time when package size dilates over time, 18-240ms and 5-90ms respectively. When more devices join the collaboration, messages dispatching over the network significantly degrades the performance from 10-15 times slower (Figure 6.12-2).

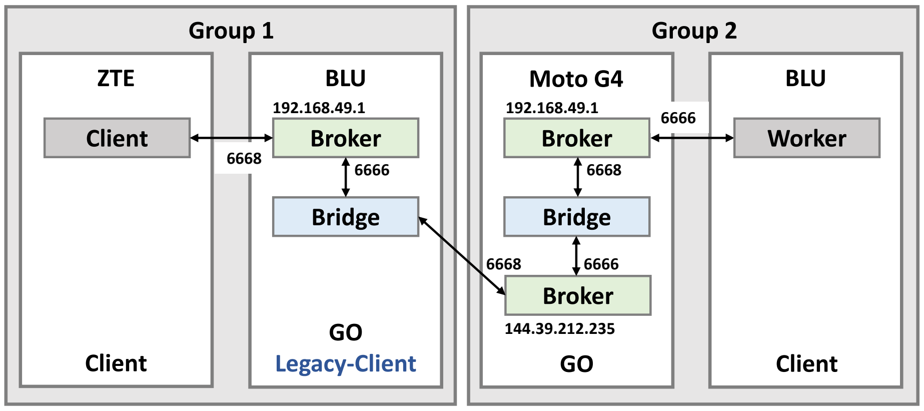


Fig. 6.13: Architecture of multi-group device-to-device communication with 4 devices.

6.5.2 Group-to-Group Communications

The flexibility of the constituent components of the our middleware make it easy and feasible for any group-to-group architectures. Figure 6.14 describes an overall example which involves multiple mobile groups, even stationary servers, where a device from one group (*P2P Client* or GO) reaches out to the GO of another group through its `Bridge`. This architecture can extend the connection range of WiFi-Direct to unlimited without any central WiFi access point because the inter-group connections are formed only by the GO devices, they are actually the virtual mobile access points for the devices from other groups to connect.

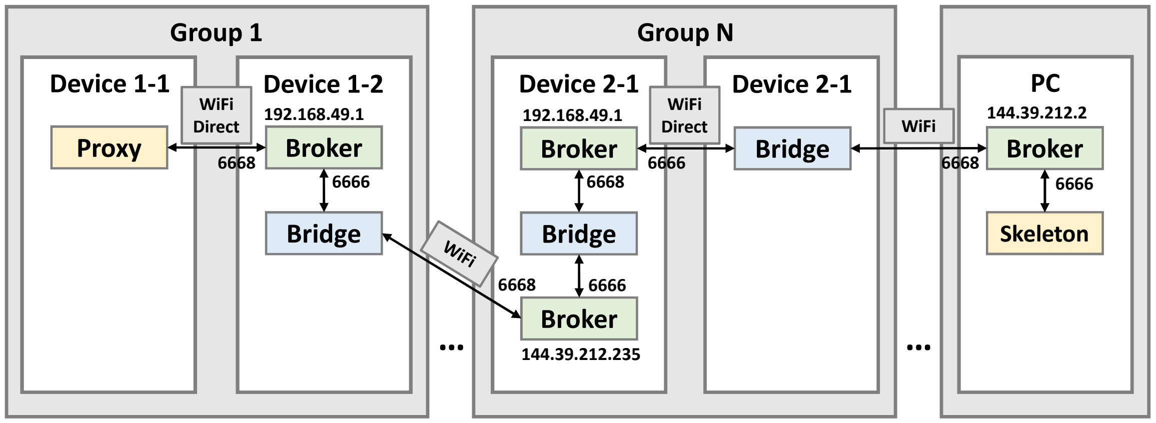


Fig. 6.14: An example of multiple groups.

Next, in the following experiments we will measure performance of our middleware when it establishes connections between mobile group(s) and PC, on multiple mobile groups and finally, INGRIM versus original RMI technology.

6.5.3 Devices To PC

An arbitrary device in group may by chance be connectible with a stationary server, which enriches the group with more powerful resources. To make the server available, one Broker will be installed there along with BackEnd(s) to receive requests and forward responses as in Figure 6.15. The device contacting server will hold one Broker and a Bridge to forward requests from its Broker to the server's one.

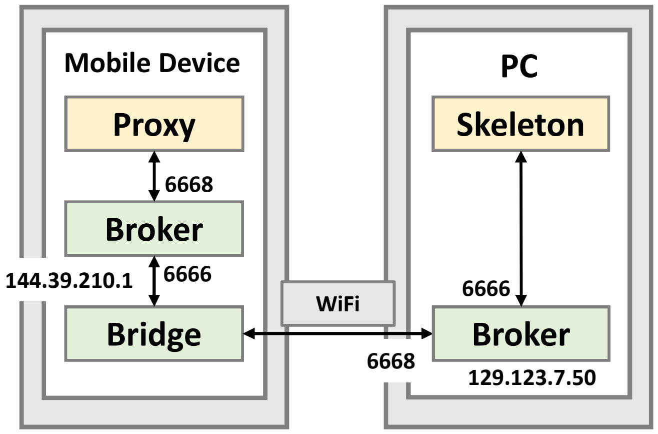
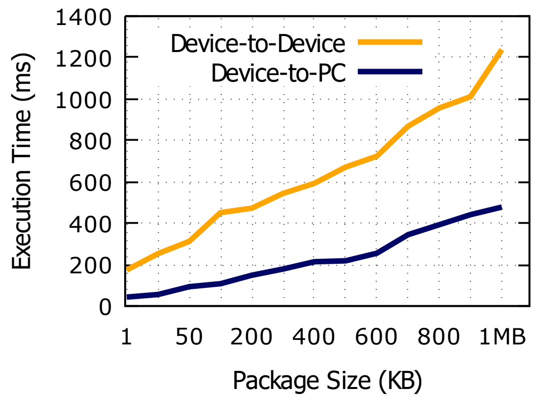


Fig. 6.15: A simple way to bridge device to PC.

Then, we compare the speed of Device(s)-to-PC over WiFi with Device-to-Device over WiFi-Direct in the same network, Figure 6.16 shows 2 cases: (1) the performance of 1 device to PC is always better varied from 2.2 to 4.5 depend on package sizes; likewise, (2) 2 devices to PC also performs 1.9-2.7 times faster.



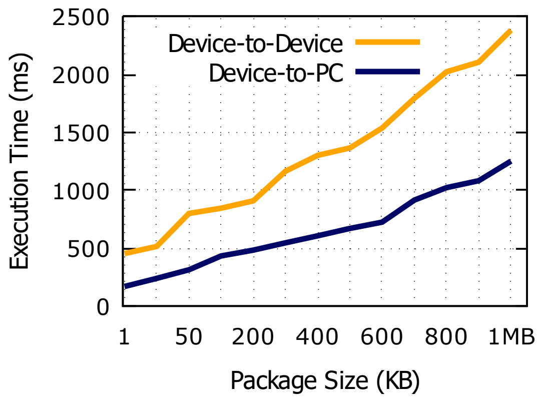


Fig. 6.16: Performance of device-to-device versus device-to-PC.

6.5.4 INGRIM vs. RMI

Android platform has limited APIs and does not support RMI technology, therefore we proceed this comparison on server environment where two servers periodically execute remote function calls on top of each platform. This evaluation relies on $T_{[Total]}$ values measured on FrontEnd for 4 different tasks: (1) sending packages with *empty function* (does nothing but returns the package size) and gradually-increasing data sizes, (2) blurring an image, (3) detecting motions in two images using OpenCV [6] and (4) counting the most frequent words in a document.

[6]OpenCV for Java: http://opencv-java-tutorials.readthedocs.io

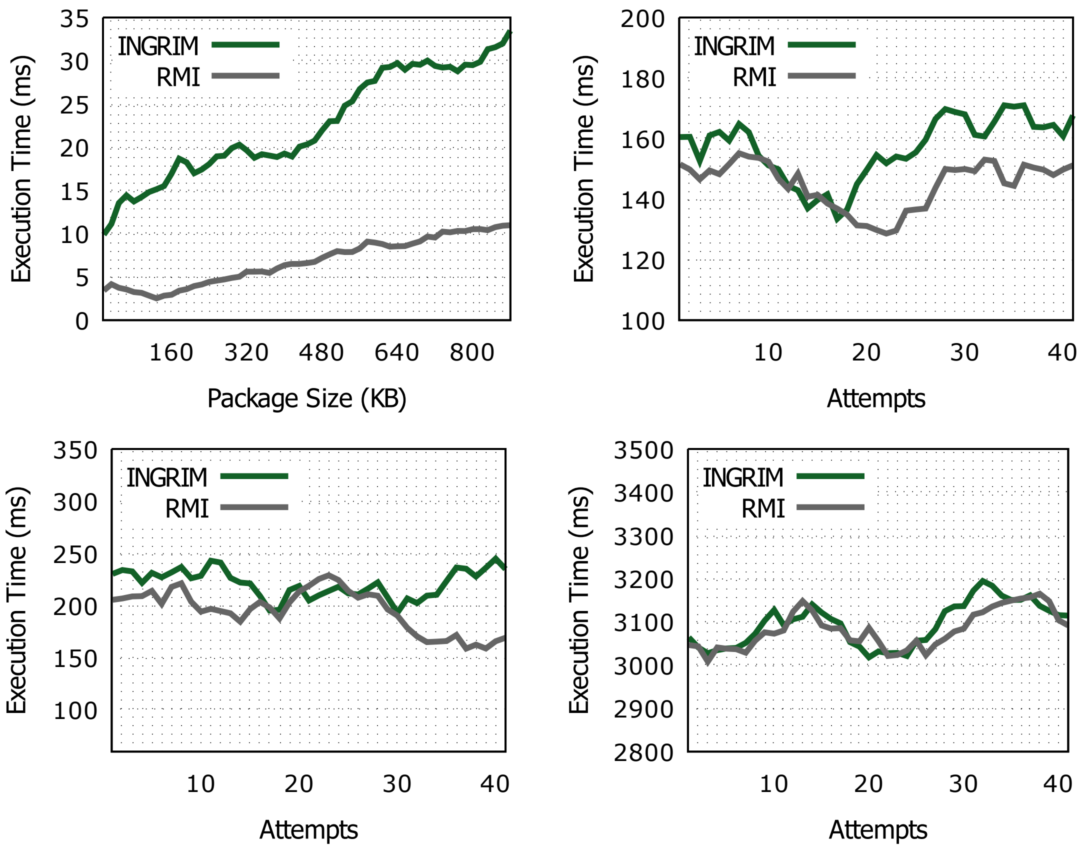


Fig. 6.17: Comparing INGRIM and RMI in 4 test cases.

Figure 6.17 depicts the different between the two technologies. In the first test case, since the empty function returns result immediately so the $T_{[Total]}$ is accumulated by mostly the network time and system overhead which is insignificant, 10-35ms by INGRIM and 3-10ms by RMI for package sizes from 1K to 1MB. In general, RMI has the less overhead which performs 70.8% faster than INGRIM. In the next two image processing test cases, the image process takes approximately 100-200ms which is 4-5 times more than their overheads, the impact of overheads becomes trivial. The results of 40 attempts (Figure 6.17-2 and 6.17-3) show slightly better performance of RMI compared with INGRIM: 7.4% better for the image blurring and 11.2% for the motion detection.

Regarding the word counting service, the average time to examine each 1MB document takes 2000-2500ms which is 100 times of the system overheads and literally makes them futile. The results of 40 attempts of word counting show the RMI only dominates INGRIM by 0.4% as their performance is almost the same. According to the results, it's obvious that our technology generates a bit more overhead than RMI but gives the similar performance in actual test cases, especially for the intensive tasks.

6.5.5 Remote Motion Detection

In this experiment one device calls remote function to detect motions of an object from the two consecutive images, the function returns an Integer array of the rectangle areas where the motions take place. To this end, we used OpenCV for Android [7] and to reduce network throughput the app reduces the images size to 180×135 and converts it to black and white before sending them out.

In the first test, we use one device to alternately connect to one of the three others, Figure 6.18-Left reveals among them Moto-G4 gives the best performance at average of 416 milliseconds per call, or 2.4fps. BLU runs 6.5% slower and ZTE 21% slower than Moto-G4.

In the second test, we compare the system performance of motion detection remote calls by 2 and 3-hop networks with the devices in the prior test. Figures 6.18-Right shows the average time for each call in 3-hop network is 885.5ms which is 55% slower than the 2-hop.

6.6 Related Work

In an effort to enable advanced RMI on mobile device, INGRIM serializes a function call into binary stream and dispatch over the networks. A different approach, Android RMI [31] leverages the *original Binder* to allow users to invoke system services as well as application services between devices using remote parcel format. Lin et al. introduces an cross-platform IPC mechanism called XBinder [32] to enable remote process among multi-user communication for mobile applications to cooperate with local or remote services without developing complicate network. However, despite the remarkable improvements

[7] OpenCV for Android: https://opencv.org/platforms/android/

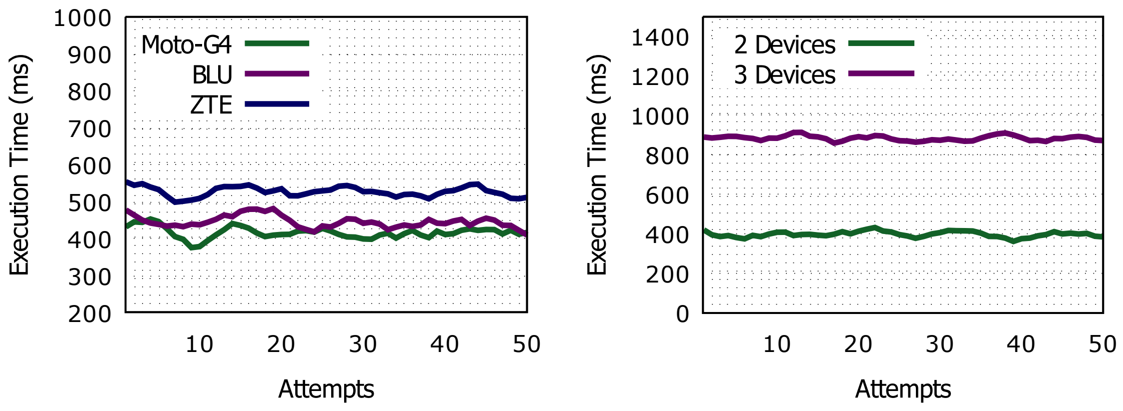


Fig. 6.18: Performance of device-to-device versus device-to-PC.

with respect to performance, these designs only aim to the mobile devices that are in the same WiFi network.

Nakao et al. [124] provides an RPC-based invocation mechanism between Android devices using *Intent*, a message format used by Android platform to realize transparent remote service communication to other devices without any modifications to the current Android applications. Similarly, Nagahara et al. [125] proposed a *distributed intent* framework where Android applications collaborate with embedded devices by sending serialized Intent messages through the network. Another approach for mobile remote process is making services public so the other devices may invoke services using remote call mechanisms [126, 127], however, this approach incurs too much overhead for the host devices as well as risks of service unavailability.

Our middleware makes contribution to the domain of WiFi-Direct multi-group communication where a group connects to another one using *legacy client*, a module operates on the original WiFi interface to serve as a bridge between the two group owners [123, 128]. Casetti et al. [29] leverages WiFi-Direct multiple groups for content-centric routing network where data is transparently available to users using *content routing tables* which collect and transport data over the content nodes, the routing tables are advertised and populated via a registration/advertisement protocol. INGRIM extends the idea of content-centric to bring function-centric architecture to the mobile network, any device can request for a function call regardless of knowing actual location of the function or whether it hosts on a mobile device or stationary server.

Finally, we share the vision in multi-group WiFi-Direct simulation since the cost for the experiment deployment is expensive as well as the discovery and network handshake phases are complex and time consuming. WiDiSi is a dedicated visual simulation [129, 130] extending PeerSim library [131] to support WiFi-Direct, it can simulate and visualize a vast device-to-device network including discovery and network establishment of devices moving randomly within closed distances, however, the disadvantages of WiDiSi (also the weakness in PeerSim) are single-thread, less autonomy and unsupported for multiple groups. The new version MAGNET [130], a novel self-organizing middleware infrastructure that aims to provide reliable and stable P2P connectivity among large numbers of smart devices.

6.7 Conclusion

This chapter introduces INGRIM, a new Inter-group Remote Invocation Middleware architecture to enable routing remote method invocation over multiple group device-to-device networks. INGRIM modularizes its architecture by the functional components using annotations which makes it flexible to apply and adaptive with either D2D or device-to-server networks. Our empirical experiments with actual test-bed devices unveil the low overhead conduct and similar performance as RMI in reality.

CHAPTER 7
FUTURE WORK

7.1 Asynchronous WiFi-Direct Simulation

Mobile network is known as volatile and intermittent. Different from stationary server networks where connections are wired and stable, mobile networks are formed by wireless connectivity interfaces such as WiFi-Direct, Bluetooth or LTE, which mainly confront the mobility and limited distances. In the experiments, as a device is gradually leaving its group its connections with the others get more attenuated, thus the measurements performed on the group become unreliable. Another problem we met is the limited number of devices, for each test case we experienced executions with up to six devices, although the results reflected exactly our system's performance and energy consumption, it would be still helpful and more accurate if we could perform further tests with 10 to 20 more devices. Finally, every device runs a number of background applications such as system services or resource management deamons, those hidden services consume CPU, occupy large amount of memory and drain battery just as the other apps making measurements incorrect. Therefore, we believe a mobile network simulator could solve all above problems.

A WiFi-Direct simulation should be defined with the following features: (1) each virtual device has initial specification values of CPU, RAM and battery (or even sensors) randomly or by user configuration, these values must be decreased according to the time or number of running applications. (2) The virtual device must be integral with our middleware so that it can run normally on as if the mobile platform. (3) They should be able to discover peers, elect owner and establish connections if they are in closed distance (predefined in by user configuration). (4) The device is able to move inside the virtual environment by random or predefined speed and direction, or it can be stationary. Finally, (5) the simulator must be visual, so that user can observe their movement, connectivity and amount of data transmitted over the inner virtual networks.

WiFi-Direct simulator has been discussed in some research before [129,130]. However, these research and related products rely on single thread architecture, they can't adopt external application and do not support autonomy of virtual devices [132].

7.2 Compensation Model

Code Offloading mechanism always raises the question about user privacy when they hesitate to receive a request of code execution from someone through WiFi-Direct or Bluetooth, even though they have been through an agreement process while establishing connection [133,134]. We proposed a compensation solution to address this problem using virtual coin [135], through five steps: (1) a novel algorithm to estimate the *cost of execution* by analyzing the complexity of offloading task and the amount of data attached to the task. (2) The client sends the proposal including execution code, data, estimated cost and the virtual coins with value equivalent to the estimated cost to broker, (3) the broker will split the proposal and cost and forward each part to the peers [136]. (4) A peer will consider the cost and send the answer back to the broker. (5) If all the answers are positive, the broker will forward the sub task and partial virtual coins to each peer, wait and collect results, then aggregate the results and push back to the client. If at least one answer is negative the broker will repeat the request to the remaining peers. In the second time if one peer returns a negative answer, the broker will consider the task as failed and push a denial message back to the client.

7.3 Autonomous Reformation

Unlike the other server networks which operate on wired mesh topology, WiFi-Direct is a wireless network formed up by the mobile devices come into single group with one device is elected as group owner. The group owner takes role of a mobile server, its DHCP service allocates IPs to the other members of the group and coordinates data transmission among the devices. If the group suddenly drops connection, the entire group will also crumble [137], to recover the connectivity, the remaining devices have to restart the discovery and election phases again which consume time a lot.

To address this problem, a solution proposed using *Additional GO* (AGO) [138]. During the GO negotiation phase [1], both devices will attach 1 bit of *Additional GO Intent* to the existing GO Intent to indicate the remain device will probably take place when the main GO leaves group. When new device joins the group, the main GO will inform the address of the substitute owner so that the clients know where it should connect to in the worst case. By using *Additional GO*, we can prohibit the discovery and negotiation phases which is to reduce time for group recovery.

[1] WiFi-Direct: `http://www.thinktube.com/tech/android/wifi-direct`

CHAPTER 8
CONCLUSION

This dissertation raised and solved five critical technical problems that are currently interfering with the applicability of inter-group device-to-device communication to improve performance and preserve energy in mobile industry. First, we proposed a novel middleware architecture on WiFi-Direct which implements a selection and division algorithm to split and distribute a task fairly to all nearby ones, based on their resource availability at the moment of making requests. The system, dependent on the number of devices, significantly improved the performance to 70% and preserved energy to 50%. Second, in highly dynamic and volatile edge computing environment, we provide a service infrastructure for reliable and efficient mobile edge computing which includes adaptive facilities to dynamically restructure the patterns of distributed communication in response to partial failure. It flexibly switches all current executing tasks to any local mobile networks in its range if communication to the edge server has failed, and restores the connection when network is recovered. Third, for platform heterogeneity, we evaluated several JavaScript engines in the market nowadays and integrated the highest performance one to our middleware system, then enabled options for developer to create execution package to proceed on either native or JavaScript engine. Forth, we overcome the limitations of device-to-device networks by providing method invocation routing infrastructure over inter-group mobile networks, the system based on constituent out-of-the box components that is highly compatible with multiple network topologies, and tolerant for network malfunctions. By extending remote function calls on mobile platforms, we were capable of deploying RMI technology not only on Android OS but extensible for Windows Phone and iOS, it also remarkably meliorated the performance of dispatching requests to meet the speed requirements of mobile real-time applications. Finally, we extended the use of mobile method invocation for effective code distribution and offloading over multi-group device-to-device networks.

In the future, we will continue the research by extending our architecture in three different directions: mobile network simulation, compensation model for collaboration and autonomous recovery for network failures. This dissertation is written based on five conference papers, sequentially presented at SAC 2017 [5], FMEC 2017 [6], COMPSAC 2017 [7], iiWAS 2017 [8] and a submitted paper to MOBILESoft 2018.

Bibliography

[1] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: Enabling remote computing among intermittently connected mobile devices," ser. MobiHoc 2012.

[2] A. A. Shahin and M. Younis, "Efficient multi-group formation and communication protocol for wi-fi direct," in *2015 IEEE 40th Conference on Local Computer Networks (LCN)*, 2015.

[3] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[4] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *MobiSys*, 2010.

[5] M. Le and Y.-W. Kwon, "Utilizing nearby computing resources for resource-limited mobile devices," in *32nd ACM SIGAPP Symposium on Applied Computing (SAC).*

[6] M. Le, Z. Song, Y. Kwon, and E. Tilevich, "Reliable and efficient mobile edge computing in highly dynamic and volatile environments," in *FMEC 2017.*

[7] M. Le, M. Song, and Y. W. Kwon, "Enabling flexible and efficient remote execution in opportunistic networks through message-oriented middleware," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, 2017.

[8] M. Le and S. W. Clyde, "Jsrex: An efficient javascript-based middleware for multi-platform mobile peer-to-peer networks," in *iiWAS 2017*, 2017.

[9] W. Y. Li, S. Wu, W. K. Chan, and T. Tse, "Jscloud: Toward remote execution of javascript code on handheld devices." in *QSIC*, 2012.

[10] G. Kulkarni, R. Shelke, R. Palwe, V. Solanke, S. Belsare, and S. Mohite, "Mobile cloud computing - bring your own device," in *2014 Fourth International Conference on Communication Systems and Network Technologies*, 2014.

[11] I. Foster, "Globus online: Accelerating and democratizing science through cloud-based services," *IEEE Internet Computing*, 2011.

[12] N. Nikzad, M. Radi, O. Chipara, and W. G. Griswold, "Managing the energy-delay tradeoff in mobile applications with tempus," in *Proceedings of the 16th Annual Middleware Conference*, 2015.

[13] G. Calice, A. Mtibaa, R. Beraldi, and H. Alnuweiri, "Mobile-to-mobile opportunistic task splitting and offloading," ser. WiMob 2015.

[14] Y.-W. Kwon and E. Tilevich, "Constraint-driven dynamic adaptation of mobile applications for quality of service," in *Mobile Computing, Applications and Services (MobiCASE), 2014 6th International Conference on.* IEEE, 2014, pp. 143–152.

[15] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, "Device-to-device communications with wi-fi direct: overview and experimentation," *Wireless Communications, IEEE*, vol. 20, no. 3, pp. 96–104, 2013.

[16] K. H. Jung, Y. Qi, C. Yu, and Y. J. Suh, "Energy efficient wifi tethering on a smartphone," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014.

[17] D. R. Choffnes and F. E. Bustamante, "Taming the torrent: a practical approach to reducing cross-isp traffic in peer-to-peer systems," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 363–374.

[18] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, 2016.

[19] Y. Xiao, P. Hui, P. Savolainen, and A. Ylä-Jääski, "Cascap: Cloud-assisted context-aware power management for mobile devices," in *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, ser. MCS '11, 2011.

[20] A. Saarinen, M. Siekkinen, Y. Xiao, J. K. Nurminen, M. Kemppainen, and P. Hui, "Smartdiet: Offloading popular apps to save energy," *SIGCOMM Comput. Commun. Rev.*, 2012.

[21] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, 2016.

[22] X. Wang, X. Liu, Y. Zhang, and G. Huang, "Migration and execution of javascript applications between mobile devices and cloud," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12, 2012.

[23] Y. Ma, X. Liu, S. Zhang, R. Xiang, Y. Liu, and T. Xie, "Measurement and analysis of mobile web cache performance," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, 2015.

[24] T.-L. Tseng, S.-H. Hung, and C.-H. Tu, "Migratom.Js: A JavaScript migration framework for distributed Web computing and mobile devices," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015.

[25] M. Yu, G. Huang, X. Wang, Y. Zhang, and X. Chen, "Javascript offloading for web applications in mobile-cloud computing," in *IEEE MS 2015*, June 2015.

[26] A. Reiter and T. Zefferer, "Flexible and secure resource sharing for mobile augmentation systems," in *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, 2016.

[27] A. Asadi and V. Mancuso, "Wifi direct and lte d2d in action," in *2013 IFIP Wireless Days (WD)*, 2013.

[28] C. Funai, C. Tapparello, and W. Heinzelman, “Enabling multi-hop ad hoc networks through wifi direct multi-group networking,” in *2017 International Conference on Computing, Networking and Communications (ICNC)*, 2017.

[29] C. Casetti, C. F. Chiasserini, L. C. Pelle, C. D. Valle, Y. Duan, and P. Giaccone, “Content-centric routing in wi-fi direct multi-group networks,” in *2015 IEEE 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, June 2015.

[30] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, “A framework for partitioning and execution of data stream applications in mobile cloud computing,” *SIGMETRICS Perform. Eval. Rev.*

[31] H. Kang, K. Jeong, K. Lee, S. Park, and Y. Kim, “Android rmi: a user-level remote method invocation mechanism between android devices,” *The Journal of Supercomputing*, 2016.

[32] T. Y. Lin, J. Chen, and J. H. Liu, “Enabling cooperative computing for android-based mobile platforms,” in *International Symposium on Computer, Consumer and Control*, 2016.

[33] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, “A survey of computation offloading for mobile systems,” *Mobile Networks and Applications*, Feb 2013.

[34] Y. W. Kwon and E. Tilevich, “Reducing the energy consumption of mobile applications behind the scenes,” in *2013 IEEE International Conference on Software Maintenance*, 2013.

[35] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: Elastic execution between mobile device and cloud,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11, 2011.

[36] A. Pathak, Y. C. Hu, and M. Zhang, “Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof,” in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys ’12, 2012.

[37] N. Fernando, S. W. Loke, and W. Rahayu, “Mobile cloud computing: A survey,” *Future Generation Computer Systems*, 2013.

[38] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, “Secure ranked keyword search over encrypted cloud data,” in *2010 IEEE 30th International Conference on Distributed Computing Systems*, 2010.

[39] T. Yan, V. Kumar, and D. Ganesan, “Crowdsearch: Exploiting crowds for accurate real-time image search on mobile phones,” in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’10, 2010.

[40] M. Satyanarayanan, “Cloudlets: At the leading edge of cloud-mobile convergence,” in *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, ser. QoSA ’13, 2013.

[41] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, “Haskell in green land: Analyzing the energy behavior of a purely functional language,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[42] T. Nabi, P. Mittal, P. Azimi, D. Dig, and E. Tilevich, “Assessing the benefits of computational offloading in mobile-cloud applications,” in *Proceedings of the 3^{rd} International Workshop on Mobile Development Lifecycle*, 2015.

[43] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong, “Rio: a system solution for sharing i/o between mobile systems,” in *Proceedings of the 12^{th} Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[44] N. Zhang, Y. Lee, M. Radhakrishnan, and R. K. Balan, “Gameon: P2P gaming on public transport,” in *MobiSys 2015.*

[45] A. J. Khan, K. Jayarajah, D. Han, A. Misra, R. Balan, and S. Seshan, “Cameo: A middleware for mobile advertisement delivery,” in *Proceedings of the 11^{th} Annual International Conference on Mobile Systems, Applications, and Services*, 2013.

[46] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, “Scalable crowd-sourcing of video from mobile devices,” in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’13, 2013.

[47] W. Emmerich, C. Mascolo, and A. Finkelsteiin, “Implementing incremental code migration with xml,” in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE ’00, 2000.

[48] I. Satoh, “Mobilespaces: a framework for building adaptive distributed applications using a hierarchical mobile agent system,” in *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, 2000.

[49] S. Fünfrocken, “Transparent migration of java-based mobile agents: Capturing and re-establishing the state of java programs,” *Personal Technologies*, 1998.

[50] A. Moshchuk, S. D. Gribble, and H. M. Levy, “Flashproxy: Transparently enabling rich web content via remote execution,” in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’08, 2008.

[51] M. Yoshida and K. Sakamoto, “Code migration control in large scale loosely coupled distributed systems,” in *Procs. of the 4th International Conf. on Mobile Technology, Applications, and Systems*, 2007.

[52] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, “Portable support for transparent thread migration in Java,” in *Proceedings of the 2^{nd} International Symposium on Agent Systems and Applications and 4^{th} International Symposium on Mobile Agents*, 2000.

[53] H. Flores and S. 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*, ser. MobiSys '13, 2013.

[54] Y. Duan, M. Zhang, H. Yin, and Y. Tang, "Privacy-preserving offloading of mobile app to the public cloud," in *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'15, 2015.

[55] H. Eom, P. S. Juste, R. Figueiredo, O. Tickoo, R. Illikkal, and R. Iyer, "Machine learning-based runtime scheduler for mobile offloading framework," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, ser. UCC '13, 2013.

[56] K. Zhang, Y. Mao, S. Leng, Q. Zhao, L. Li, X. Peng, L. Pan, S. Maharjan, and Y. Zhang, "Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks," *IEEE Access*, vol. 4, pp. 5896–5907, 2016.

[57] X. Sun and N. Ansari, "Primal: Profit maximization avatar placement for mobile edge computing," in *2016 IEEE International Conference on Communications (ICC)*, May 2016.

[58] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys Tutorials*, vol. 19, thirdquarter 2017.

[59] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, pp. 37–42.

[60] Y. Jararweh, A. Doulat, O. AlQudah, E. Ahmed, M. Al-Ayyoub, and E. Benkhelifa, "The future of mobile cloud computing: Integrating cloudlets and mobile edge computing," in *2016 23rd International Conference on Telecommunications (ICT)*, May 2016.

[61] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *SIGCOMM Comput. Commun. Rev.*, 2014.

[62] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, December 2016.

[63] "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges," *Future Generation Computer Systems*, vol. 78, pp. 680 – 698, 2018.

[64] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Transactions on Networking*, vol. 24, October 2016.

[65] O. Mkinen, "Streaming at the edge: Local service concepts utilizing mobile edge computing," in *2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies*, Sept 2015.

[66] A. S. Maciej Rostanski, Krzysztof Grochla, “Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq,” 2014.

[67] H. C. Frank H. P. Fitzek, *Mobile Peer-to-peer (P2P): A Tutorial Guide*, ser. Wiley, 2009.

[68] R. M. P. Bellavista, A. Corradi and C. Stefanelli, “Context-aware middleware for resource management in the wireless internet,” in *IEEE Transactions on Software Engineering.* IEEE, 2003, pp. 1086–1099.

[69] L. McNamara, C. Mascolo, and L. Capra, “Media sharing based on colocation prediction in urban transport,” in *Proceedings of the 14*th *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2008.

[70] C. H. Yu, K. Doppler, C. B. Ribeiro, and O. Tirkkonen, “Resource sharing optimization for device-to-device communication underlaying cellular networks,” *IEEE Transactions on Wireless Communications*, vol. 10, no. 8, pp. 2752–2763, 2011.

[71] F. Cuomo, C. Martello, A. Baiocchi, and F. Capriotti, “Radio resource sharing for ad hoc networking with uwb,” *IEEE J.Sel. A. Commun.*, vol. 20, pp. 1722–1732, Sep. 2006.

[72] W. Emmerich, C. Mascolo, and A. Finkelstein, “Implementing incremental code migration with XML,” in *Proceedings of the 22*nd *International Conference on Software Engineering*, 2000.

[73] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, “Offload – automating code migration to heterogeneous multicore systems,” in *Proceedings of the 5*th *International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[74] Y.-W. Kwon and E. Tilevich, “Energy-efficient and fault-tolerant distributed mobile execution,” in *Proceedings of the 32*nd *International Conference on Distributed Computing Systems (ICDCS ’12)*, June 2012.

[75] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong, “DR-OSGi: Hardening distributed components with network volatility resiliency,” in *ACM/IFIP/USENIX Middleware 2009.*

[76] Y.-W. Kwon and E. Tilevich, “Configurable and adaptive middleware for energy-efficient distributed mobile computing,” ser. MobiCASE ’14.

[77] Wi-Fi Alliance, “Wi-Fi certified Wi-Fi Direct,” 2010.

[78] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” 2003.

[79] P. Hintjens, *ZeroMQ: Messaging for Many Applications.* O’Reilly Media, Inc., 2013.

[80] R. Henjes, D. Schlosser, M. Menth, and V. Himmler, “Throughput performance of the activemq jms server.” Spr. Berlin Heidelberg ’07.

[81] G. Kalic, I. Bojic, and M. Kusek, “Energy consumption in android phones when using wireless communication technologies,” in *2012 Proceedings of the 35th International Convention MIPRO*, 2012.

[82] L. Jiao, R. Friedman, X. Fu, S. Secci, Z. Smoreda, and H. Tschofenig, “Cloud-based computation offloading for mobile devices: State of the art, challenges and opportunities,” in *2013 Future Network Mobile Summit*, July 2013.

[83] S. Tilak, P. Hubbard, M. Miller, and T. Fountain, “The ring buffer network bus (rbnb) dataturbine streaming data middleware for environmental observing systems,” ser. e-Science and Grid Computing, 2007.

[84] O. Helgason, S. T. Kouyoumdjieva, L. Pajevic, E. A. Yavuz, and G. Karlsson, “A middleware for opportunistic content distribution.”

[85] Y. Wu, M. Barnard, and L. Ying, “Architecture and implementation of an information-centric device-to-device network,” ser. 53rd Allerton, 2015.

[86] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, “Cosmos: Computation offloading as a service for mobile devices,” ser. MobiHoc ’14.

[87] P. Costa, C. Mascolo, M. Musolesi, and G. P. Picco, “Socially-aware routing for publish-subscribe in delay-tolerant mobile ad hoc networks,” *IEEE Journal on Selected Areas in Communications*, 2008.

[88] X. Tong and E. C. H. Ngai, “A ubiquitous publish/subscribe platform for wireless sensor networks with mobile mules,” in *2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems*, 2012.

[89] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, “Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges,” *IEEE Communications Surveys Tutorials*, 2014.

[90] P. H. Sokol Kosta, Andrius Aucinas and R. Mortier, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *INFOCOM ’12*.

[91] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, “Mobile code offloading: from concept to practice and beyond,” *IEEE Communications Magazine*, 2015.

[92] Y. Li and W. Gao, “Code offload with least context migration in the mobile cloud,” in *INFOCOM*, 2015.

[93] C. Xu, Y. Qiao, B. Lee, and N. Murray, “Energy consumption of mobile offloading for javascript applications,” in *ISSC*, 2015.

[94] X. S. Wang, H. Shen, and D. Wetherall, “Accelerating the mobile web with selective offloading,” in *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, ser. MCC ’13, 2013.

[95] S. Park, Q. Chen, and H. Y. Yeom, "Pios: A platform-independent offloading system for a mobile web environment," in *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*, 2013.

[96] I. Hwang and J. Ham, "Cloud offloading method for web applications," in *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, April 2014.

[97] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th ACM European Conference on Computer Systems*, 2011.

[98] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring android java code for on-demand computation offloading," in *OOPSLA '12*.

[99] H. Flores and S. Srirama, "Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning," in *Proceeding of the Fourth ACM Workshop on Mobile Cloud Computing and Services*, ser. MCS '13, 2013.

[100] C. Funai, C. Tapparello, and W. B. Heinzelman, "Supporting multi-hop device-to-device networks through wifi direct multi-group networking," *CoRR*, 2016.

[101] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, 2010.

[102] D. Feng, L. Lu, Y. Yuan-Wu, G. Y. Li, S. Li, and G. Feng, "Device-to-device communications in cellular networks," *IEEE Communications Magazine*, 2014.

[103] X. Zhang, W. Jeon, S. Gibbs, and A. Kunjithapatham, "Elastic html5: Workload offloading using cloud-based web workers and storages for mobile devices," in *Mobile Computing, Applications, and Services*, M. Gris and G. Yang, Eds. Springer Berlin Heidelberg, 2012.

[104] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "Trustjs: Trusted client-side execution of javascript," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17. ACM, 2017.

[105] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the emerald system," *ACM Trans. Comput. Syst.*, 1988.

[106] X. Wang, X. Liu, G. Huang, and Y. Liu, "Appmobicloud: Improving mobile web applications by mobile-cloud convergence," in *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, 2013.

[107] M. Zbierski and P. Makosiej, "Bring the cloud to your mobile: Transparent offloading of html5 web workers," in *CloudCom*, 2014.

[108] H. J. Jeong and S. M. Moon, "Offloading of web application computations: A snapshot-based approach," in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, Oct 2015, pp. 90–97.

[109] D. Chu, Z. Zhang, A. Wolman, and N. Lane, “Prime: A framework for co-located multi-device apps,” in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp ’15.

[110] M. Shiraz and A. Gani, “A lightweight active service migration framework for computational offloading in mobile cloud computing,” *The Journal of Supercomputing*, 2014.

[111] K. Ravichandran, A. Gavrilovska, and S. Pande, “Pimico: Privacy preservation via migration in collaborative mobile clouds,” in *2015 48th Hawaii International Conference on System Sciences*, 2015.

[112] R. K. K. Ma and C. L. Wang, “Lightweight application-level task migration for mobile cloud computing,” in *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, 2012.

[113] W. Zhu, C.-L. Wang, and F. C. M. Lau, “Jessica2: a distributed java virtual machine with transparent thread migration support,” in *Proceedings. IEEE International Conference on Cluster Computing*, 2002.

[114] H. Mei and J. Lü, *Refactoring Android Java Code for On-Demand Computation Offloading.* Singapore: Springer Singapore, 2016, pp. 337–358.

[115] S. Jadhav, S. Dutia, K. Calangutkar, T. Oh, Y. H. Kim, and J. N. Kim, “Cloud-based android botnet malware detection system,” in *2015 17th International Conference on Advanced Communication Technology (ICACT)*, July 2015, pp. 347–352.

[116] S. W. Clyde and E. J., “Unifying definitions for modularity, abstraction, and encapsulation as a step toward foundational multi-paradigm software engineering principles,” in *Proceedings of the Twelve International Conference on Software Engineering Advances, Athens, Greece, Oct. 8-12*, 2017.

[117] A. Borriello, F. Melillo, and G. Canfora, “Migrating android applications towards service-centric architectures with sip2share,” in *2013 17th European Conference on Software Maintenance and Reengineering*, March 2013, pp. 413–416.

[118] J. Siegel, *Corba 3 fundamentals and programming.* John Wiley & Sons, 2000.

[119] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design.* Pearson, 2011.

[120] “Efficient java rmi for parallel programming,” *ACM Trans. Program. Lang. Syst.*, 2001.

[121] S. Adwankar, “Mobile corba,” in *Proceedings 3rd International Symposium on Distributed Objects and Applications*, 2001.

[122] S. Trifunovic, B. Distl, D. Schatzmann, and F. Legendre, “Wifi-opp: Ad-hoc-less opportunistic networking,” in *Proceedings of the 6th ACM Workshop on Challenged Networks*, 2011.

[123] C. Funai, C. Tapparello, and W. Heinzelman, "Supporting multi-hop device-to-device networks through wifi direct multi-group networking," in *Transactions on Computing Research Repository*, 2016.

[124] K. Nakao and Y. Nakamoto, "Toward remote service invocation in android," in *2012 9th UIC/ATC*, 2012.

[125] Y. Nagahara, H. Oyama, T. Azumi, and N. Nishio, "Distributed intent: Android framework for networked devices operation," in *IEEE CSE 16th*, 2013.

[126] M. Toyama, S. Kurumatani, J. Heo, K. Terada, and E. Y. Chen, "Android as a server platform," in *2011 IEEE CCNC*.

[127] J. Choi and J. Park, "A framework for remote service invocation of android services to communicate with external services in java environment," in *Journal of the Korea society of IT services*, 2013.

[128] A. Teófilo, D. Remédios, H. Paulino, and J. a. Lourenço, "Group-to-group bidirectional wi-fi direct communication with two relay nodes," in *MOBIQUITOUS*, 2015.

[129] L. Baresi, N. Derakhshan, and S. Guinea, "Widisi: A wi-fi direct simulator," in *2016 IEEE Wireless Communications and Networking Conference*, April 2016, pp. 1–7.

[130] L. Baresi, N. Derakhshan, S. Guinea, and F. Arenella, "Magnet: A middleware for the proximal interaction of devices based on wi-fi direct," in *IEEE ICC*, 2017.

[131] A. Montresor and M. Jelasity, "Peersim: A scalable p2p simulator," in *IEEE 9th International Conference on Peer-to-Peer Computing*, 2009.

[132] Y. J. Lai, Y. Ng, T. Sakoda, Y. Bando, A. Miyamoto, M. Ishiyama, K. i. Maeda, and Y. Doi, "Real and simulator testbeds for content dissemination in high-density large-scale wanet," in *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Jan 2017, pp. 810–817.

[133] M. Maier, "Fiwi access networks: Future research challenges and moonshot perspectives," in *2014 IEEE International Conference on Communications Workshops (ICC)*, June 2014, pp. 371–375.

[134] C. Liu, L. Zhang, M. Zhu, J. Wang, L. Cheng, and G. K. Chang, "A novel multi-service small-cell cloud radio access network for mobile backhaul and computing based on radio-over-fiber technologies," *Journal of Lightwave Technology*, 2013.

[135] P. Lalos, A. Korres, C. K. Datsikas, G. S. Tombras, and K. Peppas, "A framework for dynamic car and taxi pools with the use of positioning systems," in *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, Nov 2009, pp. 385–391.

[136] Y. Wang, H. Zhang, C. C. Tan, X. Du, and B. Sheng, "Wigroup: A lightweight cellular-assisted device-to-device network formation framework," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf*

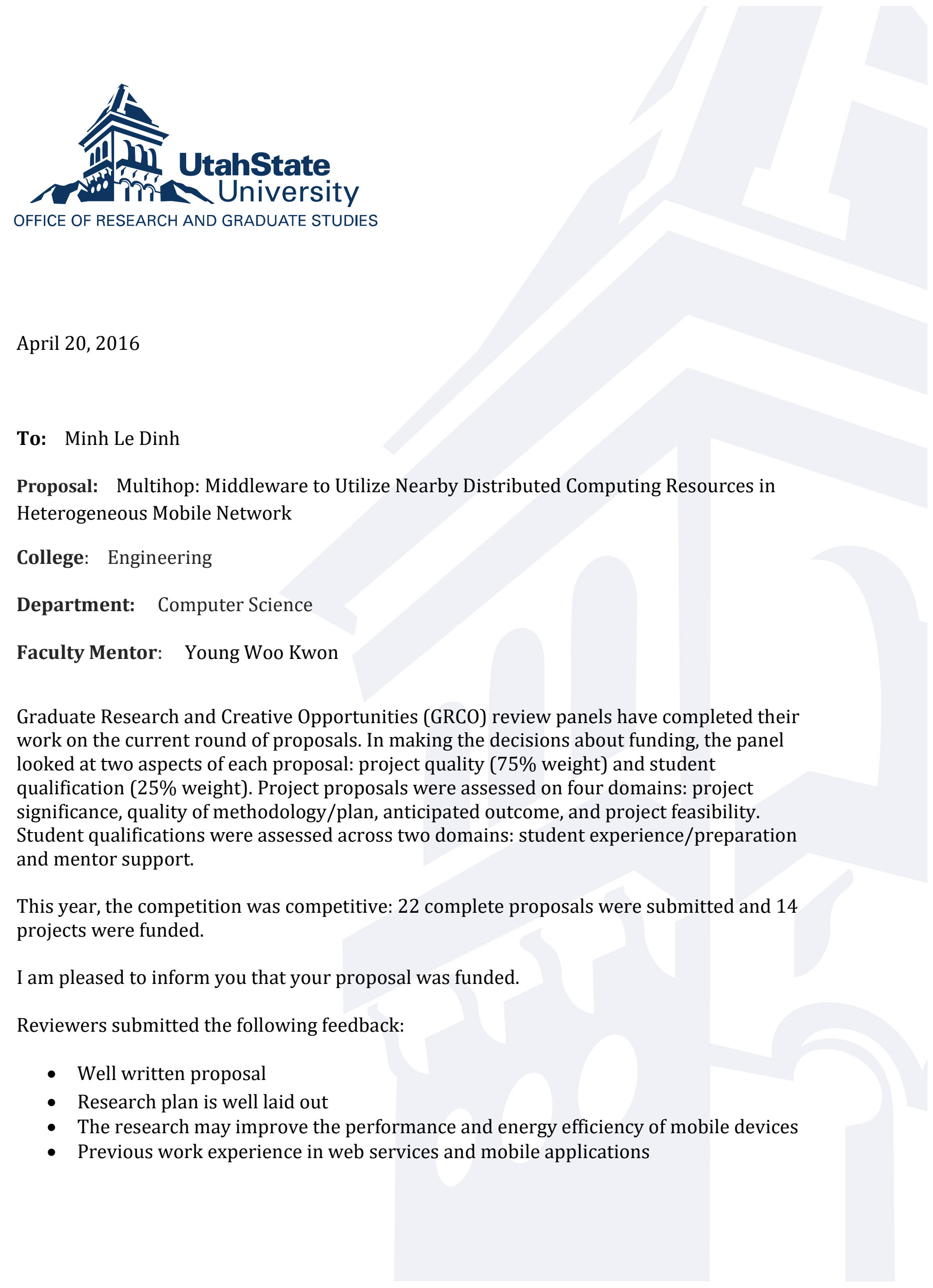
on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Aug 2015, pp. 292–299.

[137] Z. Mao, J. Ma, Y. Jiang, and B. Yao, “Performance evaluation of wifi direct for data dissemination in mobile social networks,” in *2017 IEEE Symposium on Computers and Communications (ISCC)*, July 2017, pp. 1213–1218.

[138] P. Chaki, M. Yasuda, and N. Fujita, “Seamless group reformation in wifi peer to peer network using dormant backend links,” in *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, 2015.

APPENDIX

APPENDIX A
GRADUATE RESEARCH AND CREATIVE OPPORTUNITIES



UtahState University
OFFICE OF RESEARCH AND GRADUATE STUDIES

April 20, 2016

To: Minh Le Dinh

Proposal: Multihop: Middleware to Utilize Nearby Distributed Computing Resources in Heterogeneous Mobile Network

College: Engineering

Department: Computer Science

Faculty Mentor: Young Woo Kwon

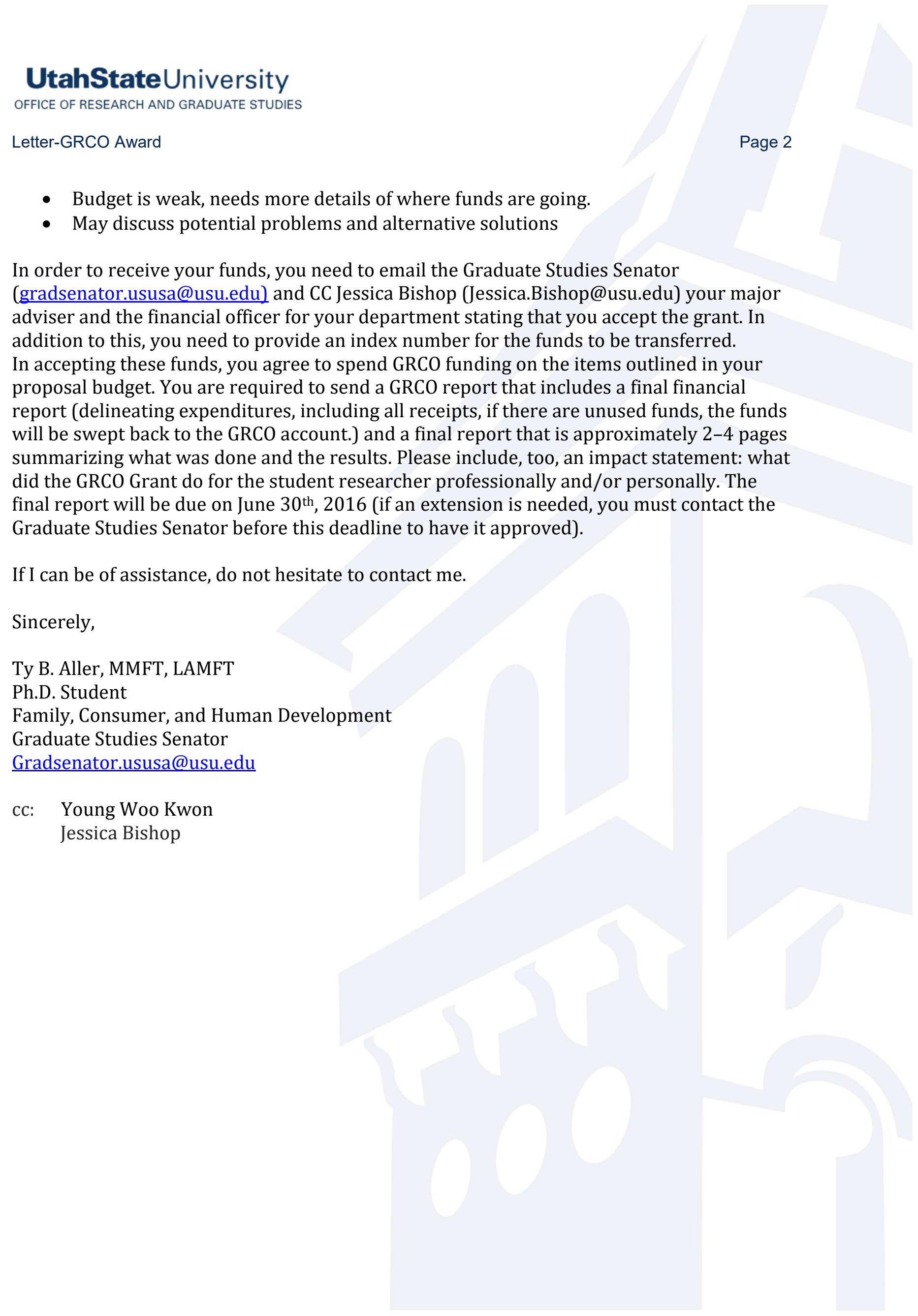
Graduate Research and Creative Opportunities (GRCO) review panels have completed their work on the current round of proposals. In making the decisions about funding, the panel looked at two aspects of each proposal: project quality (75% weight) and student qualification (25% weight). Project proposals were assessed on four domains: project significance, quality of methodology/plan, anticipated outcome, and project feasibility. Student qualifications were assessed across two domains: student experience/preparation and mentor support.

This year, the competition was competitive: 22 complete proposals were submitted and 14 projects were funded.

I am pleased to inform you that your proposal was funded.

Reviewers submitted the following feedback:

- Well written proposal
- Research plan is well laid out
- The research may improve the performance and energy efficiency of mobile devices
- Previous work experience in web services and mobile applications



UtahStateUniversity
OFFICE OF RESEARCH AND GRADUATE STUDIES

Letter-GRCO Award Page 2

- Budget is weak, needs more details of where funds are going.
- May discuss potential problems and alternative solutions

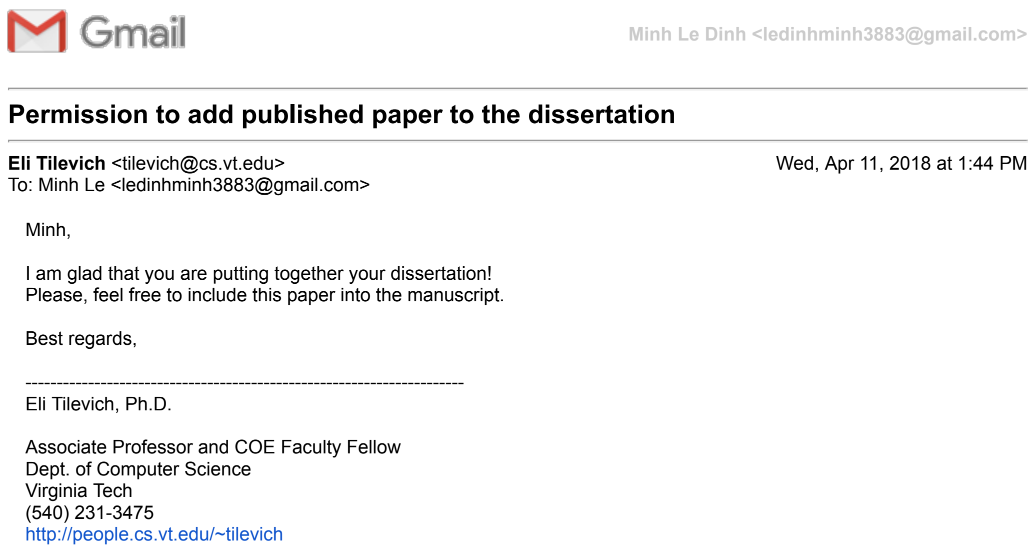
In order to receive your funds, you need to email the Graduate Studies Senator (gradsenator.ususa@usu.edu) and CC Jessica Bishop (Jessica.Bishop@usu.edu) your major adviser and the financial officer for your department stating that you accept the grant. In addition to this, you need to provide an index number for the funds to be transferred.
In accepting these funds, you agree to spend GRCO funding on the items outlined in your proposal budget. You are required to send a GRCO report that includes a final financial report (delineating expenditures, including all receipts, if there are unused funds, the funds will be swept back to the GRCO account.) and a final report that is approximately 2–4 pages summarizing what was done and the results. Please include, too, an impact statement: what did the GRCO Grant do for the student researcher professionally and/or personally. The final report will be due on June 30th, 2016 (if an extension is needed, you must contact the Graduate Studies Senator before this deadline to have it approved).

If I can be of assistance, do not hesitate to contact me.

Sincerely,

Ty B. Aller, MMFT, LAMFT
Ph.D. Student
Family, Consumer, and Human Development
Graduate Studies Senator
Gradsenator.ususa@usu.edu

cc: Young Woo Kwon
Jessica Bishop



Minh Le Dinh <ledinhminh3883@gmail.com>

Permission to add published paper to the dissertation

Eli Tilevich <tilevich@cs.vt.edu> Wed, Apr 11, 2018 at 1:44 PM
To: Minh Le <ledinhminh3883@gmail.com>

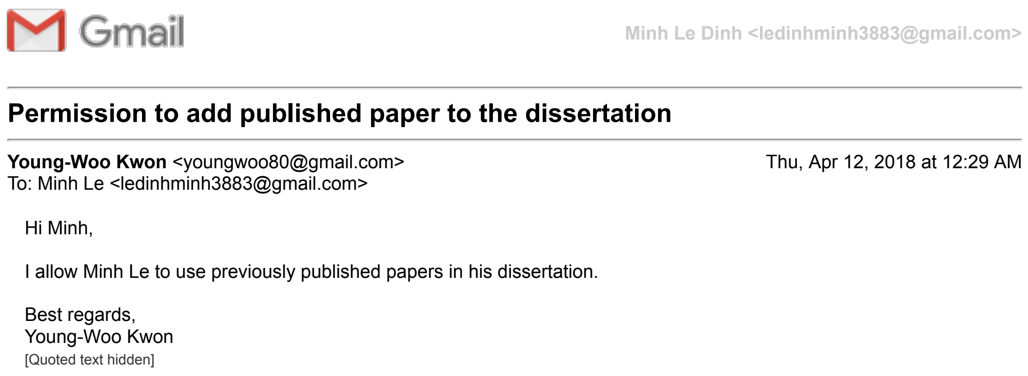
Minh,

I am glad that you are putting together your dissertation!
Please, feel free to include this paper into the manuscript.

Best regards,

Eli Tilevich, Ph.D.

Associate Professor and COE Faculty Fellow
Dept. of Computer Science
Virginia Tech
(540) 231-3475
http://people.cs.vt.edu/~tilevich



Minh Le Dinh <ledinhminh3883@gmail.com>

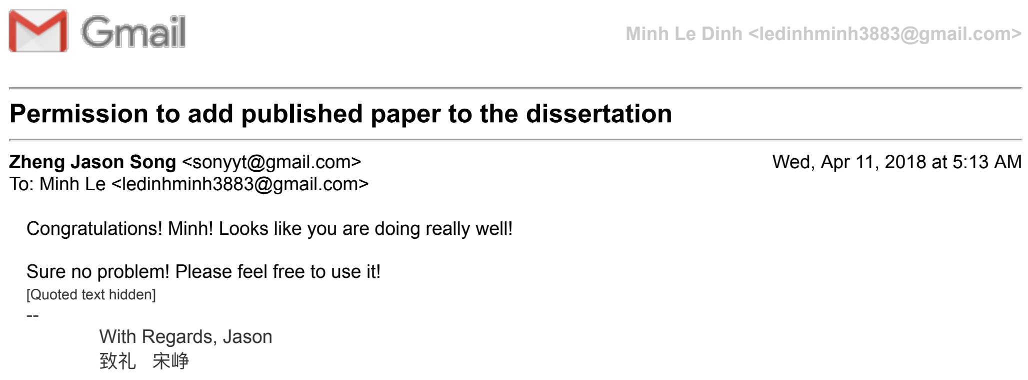
Permission to add published paper to the dissertation

Young-Woo Kwon <youngwoo80@gmail.com> Thu, Apr 12, 2018 at 12:29 AM
To: Minh Le <ledinhminh3883@gmail.com>

Hi Minh,

I allow Minh Le to use previously published papers in his dissertation.

Best regards,
Young-Woo Kwon
[Quoted text hidden]



Minh Le Dinh <ledinhminh3883@gmail.com>

Permission to add published paper to the dissertation

Zheng Jason Song <sonyyt@gmail.com> Wed, Apr 11, 2018 at 5:13 AM
To: Minh Le <ledinhminh3883@gmail.com>

Congratulations! Minh! Looks like you are doing really well!

Sure no problem! Please feel free to use it!
[Quoted text hidden]
--
With Regards, Jason
致礼　宋峥

CURRICULUM VITAE

Minh Le

Education

- **Utah State University** (2015 - 2018) *PhD* - Designed and developed an Universal Middleware for code offloading and data transmission in multiple group device-to-device collaborations. The middleware offers trust remote services based on a selection algorithm which improve performance to 70% and conserve energy to 50%.
- **Konkuk University** (2008 - 2010) *Master's Degree* - Developed SensorBridge: a system extending Message-oriented middleware to facilitate integration of a ubiquitous system to any wireless sensor networks. SensorBridge is adaptive with most of the sensor types, easy to embedded to web and 3D applications.

Working History

- **NSoft Inc** (Aug 2012 - Aug 2015) *Web and Mobile Developer* - Developed a curation and publishing system for eBook-Reader community. Developed high load balancing back-end REST services using Tomcat cluster. The services include content and storage management, document conversion and system security for web and mobile apps. Developed front-end interactive web, Android and Windows Phone applications.
- **IriTech Inc** (Jun 2010 - Jan 2012) *Enterprise Software Developer* - Designed and developed cluster-based SOAP web services wrapping iris recognition library to enable matching and security services for mobile and web applications. Developed distributed shared memory on JBoss clusters and load balancing on MySQL clusters to substantially improve template loading and comparing processes. Developed visual toolkit to

automatically detect memory leaks of the matching library (100%) and help evaluating the correctness of the implemented algorithms up to 98%.

- **FPT Information System** (Sep 2007 - Mar 2008) *Software Integration Engineer* - Developed integrated SharePoint web portals for governmental agencies. Built automatic converting tools to migrate and synchronize data from Oracle DB to Microsoft SQL Server and Business Intelligence tools which significantly speedup report generation.
- **FPT Software** (Aug 2005 - Sep 2007) *Mobile Developer* - Participated in the Taskforce team to propose solutions and solve difficult problems in the projects. Developed and maintained 6 outsourcing projects.

Published Conference Papers

- **Minh Le**, Stephen W. Clyde. *JSReX: An Efficient JavaScript-based Middleware for Multi-platform Mobile Peer-to-Peer Networks.* iiWAS 2017
- Zheng Song, **Minh Le**, Young-Woo Kwon, Eli Tilevich. *Extemporaneous Micro-Mobile Service Execution Without Code Sharing.* HotPOST 2017
- **Minh Le**, Young-Woo Kwon. *Enabling Flexible and Efficient Remote Execution in Opportunistic Networks through Message-Oriented Middleware.* COMPSAC 2017
- **Minh Le**, Zheng Song, Eli Tilevich, Young-Woo Kwon, *Reliable and Efficient Mobile Edge Computing in Highly Dynamic and Volatile Environments*, FMEC 2017
- **Minh Le**, Young-Woo Kwon, *Utilizing Nearby Computing Resources for Resource-Limited Mobile Devices*, SAC 2017
- Daehyeok Mun, **Minh Le**, Young-Woo Kwon, *An Assessment of Internet of Things Protocols for Resource-Constrained Applications*, COMPSAC 2016